TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Lehrstuhl für Datenbanken

Diplomarbeit

ÜBERBLICK UND KLASSIFIKATION LEICHTGEWICHTIGER KOMPRESSIONSVERFAHREN IM KONTEXT HAUPTSPEICHERBASIERTER DATENBANKSYSTEME

Juliana Hildebrandt

Matrikelnummer: 3278821

Betreuer

Dr.-Ing. Dirk Habich

Patrick Damme, M.Sc.

Betreuender Hochschullehrer

Prof. Dr.-Ing. Wolfgang Lehner

Eingereicht am: 24. März 2015

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit mit dem Titel *Überblick und Klassifikation leichtgewichtiger Kompressionsverfahren im Kontext hauptspeicherbasierter Datenbanksysteme* selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in der Arbeit angegebenen Hilfsmittel und Quellen benutzt. Die wörtlichen und sinngemäß übernommenen Zitate habe ich als solche kenntlich gemacht. Es waren keine weiteren Personen an der geistigen Herstellung der vorliegenden Arbeit beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 24. März 2015

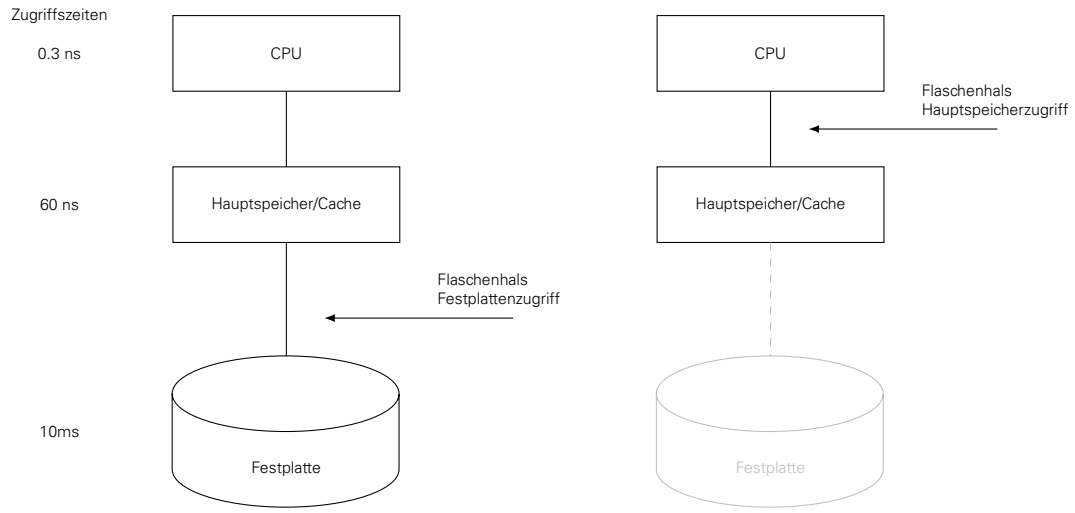
Juliana Hildebrandt

INHALTSVERZEICHNIS

1 EINLEITUNG

Klassische Datenbanksysteme verfolgen einen diskzentrischen Architekturansatz. Dabei wird, wie in Abbildung 1.1a dargestellt, die Festplatte als Datenspeicher genutzt. Beim Datenzugriff werden Seiten von der Festplatte in den Hauptspeicher geholt. Die benötigten Zeiten für Festplattenzugriffe (ca. 10 ms) und Hauptspeicherzugriffe (ca. 60 ns) unterscheiden sich um Größenordnungen. Die Datenrate zwischen CPU und Hauptspeicher ist nicht das begrenzende Element. Der Flaschenhals beim Speicherzugriff liegt deutlich beim Zugriff auf die Festplatte. Ziel diskzentrischer Architekturansätze ist es, den Zugriff auf Externspeicher über effiziente Indexstrukturen und gute Anfrageoptimierer zu verbessern.

In den letzten Jahrzehnten ging man mehr und mehr dazu über, Daten im Hauptspeicher abzulegen, um langsame Festplattenzugriffe zu vermeiden. Unter In-Memory-Datenbanksystemen versteht man spezielle Datenbanksysteme, die ihren Datenbestand im Hauptspeicher halten und keine Seiten zwischen Hauptspeicherpuffer und Festplatte tauschen. Hauptspeicherbasierte Datenbanksysteme gibt es schon länger, für spezielle Anwendungen gab es bereits vor 1990 Ansätze Daten im Hauptspeicher abzulegen. Ein älteres Beispiel ist die Entwicklung von Dali von 1993 bis 1995, später in DataBlitz umbenannt, einer In-Memory-Datenbank, deren Anwendung unter anderem in einer Echtzeitgebührenerfassung für das Telekommunikationsunternehmen Lucent Technologies bestand. Mit dem Fortschritt im Hardwarebereich, der die Bezahlbarkeit größerer Arbeitsspeicher mit sich brachte, gewannen hauptspeicherbasierte Datenbanksysteme im kommerziellen und wissenschaftlichen Bereich immer mehr an Bedeutung. Da der Zugriff auf den Hauptspeicher sehr viel schneller ist als der Zugriff auf die Festplatte, bietet sich ihre Nutzung an, wenn performancekritische Operationen ausgeführt werden müssen oder einfach sehr viele Daten in Echtzeit verarbeitet werden sollen. Ein Beispiel aus der neueren Zeit ist die Analyse und Auswertung von Genomdaten, um Krebstherapien im Rahmen der personalisierten Medizin schneller und passender auf einen individuellen Patienten zuschneiden zu können. Für eine Genomsequenzierung fallen pro Patient mehrere Gigabyte an Daten an. Die Geschwindigkeiten, mit denen Ergebnisse ausgewertet



(a) Festplattenzugriff als Flaschenhals (b) Hauptspeicherzugriff als Flaschenhals

Abbildung 1.1: Flaschenhals beim Speicherzugriff

werden können, sind essentiell für individuelle Behandlungsentscheidungen.

Mit einem hauptspeicherzentrischen Architekturansatz, wie ihn Abb. 1.1b zeigt, entfällt der Festplattenzugriff als Flaschenhals. Hauptaugenmerk liegt beim Hauptspeicherzugriff, der einen neuen Flaschenhals darstellt. Es gilt, die interne Datenrepräsentation im Hauptspeicher zu optimieren. Da sowohl Basisdaten als auch Zwischenergebnisse auf dem gleichen Medium gespeichert sind, sind Zugriffe auf Zwischenergebnisse genauso teuer wie Zugriffe auf Basisdaten. Eine Möglichkeit den Speicherzugriff zu optimieren besteht darin, sowohl für Zwischenergebnisse als auch für Basisdaten geeignete Kompressionsverfahren einzusetzen.

Klassische Kompressionsverfahren wie die Huffman-Kodierung, LZ-Varianten etc., die eher in den Jahren von 1950 bis 1980 entwickelt wurden, erzielen hohe Kompressionsraten, aber sie sind rechenintensiv und werden deshalb unter der Kategorie der schwergewichtigen Kompressionsverfahren gefasst. Der Einsatz derartiger Verfahren bringt im Kontext von hauptspeicherbasierten Datenbanksystemen keinen Nutzen, weil die verringerten I/O-Transferzeiten für das Lesen komprimierter Daten durch die zur Dekompression benötigte Zeit wieder aufgehoben werden. Wesentlich besser geeignet für den Einsatz in Datenbanksystemen sind leichtgewichtige Kompressionsverfahren, bei denen die zusätzliche Zeit für die Dekompression verglichen mit der eingesparten I/O-Transferzeit weniger ins Gewicht fällt.

Leichtgewichtige Kompressionsverfahren benötigen oft Kontextwissen über die zu komprimierenden Daten, um ähnlich gute Kompressionsraten zu erzielen wie schwergewichtige Verfahren. Unter Kontextwissen fällt jedes im Vornherein vorhandene Wissen über Datenstrukturen, Zugriffsmuster, bekannte Wahrscheinlichkeitsverteilungen, das Vorhandensein lokaler Unterschiede zwischen Daten, bedingte Wahrscheinlichkeiten für das Auftreten von Werten und damit jedes Wissen, das die Un-

ausweichlichkeit der Annahme von Gleichverteilungen, d.h. von Zufällen, aufhebt. Desweiteren gehören natürlich auch Informationen zu Hardwarespezifika wie zum Beispiel Cache-Größen dazu, so dass Datenstrukturen wie Wörterbücher mit dazu passender Größe berechnet werden können.

Ziel ist es, mit vorhandenem Kontextwissen automatisiert eine Wahl für Kompressionsverfahren zu treffen. Beispielsweise eignet sich bei sortiert vorliegenden Daten eine Differenzkodierung. Bei hohen Wahrscheinlichkeiten dafür, dass gleiche Werte oft nacheinander auftreten, bieten sich Verfahren mit Lauflängenkodierung an. Für die Auswahl eines geeigneten Kompressionsalgorithmus wird ein Repository anzulegen sein.

Eine noch bessere Anpassung an den Kontext gelingt, wenn Kompressionsverfahren modularisiert werden können. Dann können einzelne Module an den Kontext adaptiert und ausgetauscht werden. Modularisierungen werden im Allgemeinen mit verschiedenen Zielen verfolgt. Mit der Modularisierung eines Algorithmus geht die Bildung unabhängiger Einheiten sowie die Gruppierung stärker zusammenhängender Teile einher, was die Übersichtlichkeit des Gesamtsystems, im Speziellen eines möglicherweise sehr komplexen Algorithmus, erhöht. Durch Änderung bzw. Austausch einzelner Module oder auch nur eingehender Parameter können Kompressionsalgorithmen leicht variiert oder untereinander ersetzt werden. Unterstützt wird dieses Anliegen durch die mit einer Modularisierung einhergehenden Kapselung von Daten. Unterschiede und Gemeinsamkeiten verschiedener spezieller Algorithmen und allgemeinerer Ansätze lassen sich damit sehr gut miteinander vergleichen. Viele Module oder Modulgruppen tauchen in verschiedenen Algorithmen immer wieder auf und könnten bei praktischen Anwendungen, wie der Erstellung eines auf einen speziellen Kontext maßgeschneiderten Kompressionsverfahrens, dementsprechend für verschiedenste Algorithmen verwendet werden. Jede Modularisierung eröffnet die Möglichkeit einer Abstraktion komplexer Gefüge. Dadurch lassen sich allgemeinere Techniken sowie Eigenschaften von Algorithmen durch weniger spezifische Module im Kompressionsschema als Muster modellieren.

Bei der Vielzahl verschiedener Kompressionsverfahren und Daten ist die Frage, welches Verfahren sich in welchem Kontext am besten eignet bzw. wie ein für den konkreten Anwendungsfall effizient komprimierender Algorithmus arbeiten soll, nicht trivial zu beantworten. Ein Nutzer kann Kontextinformationen über seine Daten einbringen, was die Suche nach einem guten Kompressionsverfahren im Idealfall deutlich einschränkt. Dazu ist es notwendig, Unterschiede zwischen verschiedenen Verfahren auszumachen, um möglicherweise ganze Verfahrensklassen per se auszuschließen oder bestimmte Muster zu fordern.

Diese Arbeit beinhaltet eine umfassende Betrachtung von Datenkompression in diesem Kontext sowie die Erstellung eines modularisierenden Schemas (Kapitel 2), welches als Grundgerüst für verschiedene Ansätze leichtgewichtiger Kompression dient (Kapitel 3), sodass mit der genauen Definition einzelner Module sowie ein- und ausgehender Parameter konkrete Algorithmen detailliert beschrieben werden kön-

nen (Kapitel 4). Weiterhin wird der Frage nachgegangen, inwieweit sich aus weiteren Eigenschaften wie beispielsweise der Adaptivität eines Algorithmus Festlegungen für ein modularisierendes Kompressionsschema ergeben (Kapitel 5).

2 MODULARISIERUNG VON KOMPRIMIERUNGSMETHODEN

Mit der Frage nach Möglichkeiten der Modularisierung verschiedener Kompressionsalgorithmen eröffnen sich viele weitere Fragen. Wie sind Kompressionsalgorithmen aufgebaut? Welche Gemeinsamkeiten gibt es in struktureller Hinsicht? Welche in operationaler? Kann man bei der Vielzahl völlig verschiedener Algorithmen, die für die unterschiedlichsten Kontexte entwickelt wurden, für verschiedene Daten- und Redundanzarten, mit sehr heterogenen Zielsetzungen überhaupt irgendeinen sinnvollen gemeinsamen Nenner finden? Und wenn ja, gibt es dann nicht bereits eine Vielzahl von Ideen und Modellen? Worauf lässt sich ein allgemeines Modularisierungsschema für Kompressionsalgorithmen aufbauen? Wie detailreich müsste es sein? Aus welchen Komponenten müsste es bestehen? Wie könnte es trotz festgesetzter Komponenten ausreichend flexibel sein?

2.1 ZUM LITERATURSTAND

Es gibt bereits wenige und eher allgemein gehaltene Modularisierungen für die Kompression von Daten. Kontext ist dabei jedoch die Datenübertragung, weniger die Speicherung selbiger. Abbildung 2.1 veranschaulicht das moderne Paradigma der Datenkompression, welches sich in den 1980er Jahren etablierte. Von einer Quelle wird ein Datenstrom zu einem Ziel gesendet. Eine andere Möglichkeit als einen FIFO-Datenspeicher gibt es nur, wenn die Sequenz an Daten endlich ist. In diesem Schema wird aus den bisher gesendeten Daten ein Datenmodell erstellt, welches sich mit dem Hinzukommen neuer Daten meist aufgrund einer Statistik adaptiert. Auf Grundlage des aktuellen Datenmodells wird die zu sendende Nachricht kodiert. Alle kodierten Nachrichten können sequentiell wieder dekodiert werden. Auch hierbei passt sich das Datenmodell immer entsprechend der bereits empfangenen Nachrichten an.

Grundsätzlich ist bei der Modularisierung gebräuchlicher Kompressionsalgorithmen

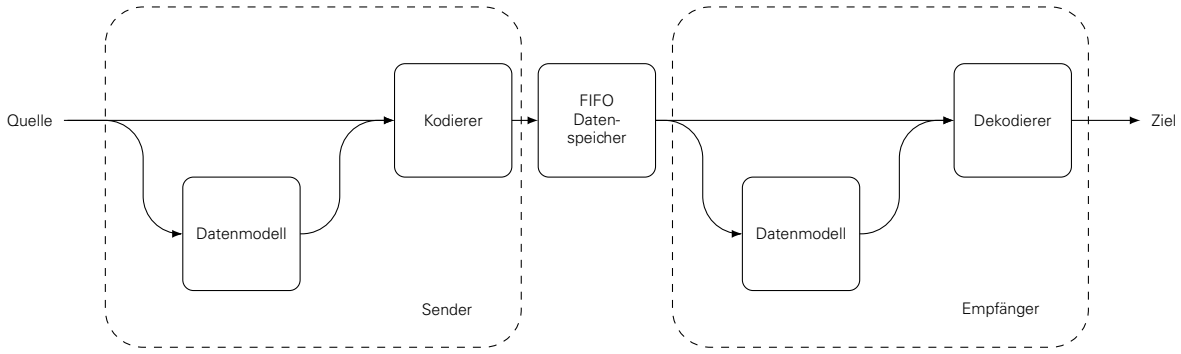


Abbildung 2.1: Modernes Paradigma der Datenkompression (nach [Wil91])

im Datenbankenkontext eine Trennung zwischen Kodierung und Modell sinnvoll. Das moderne Paradigma der Datenkompression ist dennoch nicht ausreichend allgemein, noch hat es einen sinnvoll nutzbaren Detailgrad, denn verschiedene Eigenschaften der Algorithmen, Algorithmenfamilien, Muster für Techniken und gespeicherten Datensequenzen können damit nicht modelliert werden.

Datenzerlegung Der zu kodierende Datenstrom muss in einzelne zu kodierende Einheiten unterteilt werden, sofern er nicht endlich ist. Das moderne Paradigma der Datenkompression geht davon aus, dass diese Unterteilung nicht mehr diskutiert werden muss. Zum Beispiel bei einer Lauflängenkodierung ist das aber durchaus der Fall.

Endliche und potentiell unendliche Datensequenzen Endliche Datensequenzen lassen sich anders verarbeiten als potentiell unendliche. Beispielsweise kann für endliche Sequenzen von Integerwerten eine geringere als die ursprünglich gegebene Bitweite festgelegt werden, mit der alle Daten dieser Sequenz kodiert werden können. Bei Sequenzen, die man nicht mit mehreren Pässen verarbeiten kann, ist dies nicht möglich. Viele Algorithmen im Kontext dieser Arbeit beschäftigen sich vielmehr mit der Kompression von endlichen Sequenzen, sodass keine Vorhersagen getroffen werden, weil Parameter aus einer ganzen Sequenz berechnet werden können. Endliche Sequenzen können zum großen Teil auch parallel verarbeitet werden.

Rekursive Verarbeitung Oft kommt es vor, dass von der Eingabesequenz ein endlicher Teil betrachtet wird, der wiederum in einzelne Wörter zerlegt wird. Zum Beispiel kann beim Frame of Reference für eine endliche Sequenz von Integern ein Referenzwert bestimmt werden, um danach jede Zahl einzeln relativ zum Referenzwert zu kodieren.

Reihenfolge der Daten Eng verbunden damit ist, dass die Datensätze nicht alle einzeln hintereinander gespeichert werden müssen, sondern dass die Reihenfolge in endlichen Sequenzen durchaus auch umorganisiert werden kann. Das moderne Paradigma der Datenkompression lässt hierfür keinen Spielraum.

Übertragung des Datenmodells und der Parameter Bei adaptiven Algorithmen, die das moderne Paradigma der Datenkompression beschreibt, müssen im Grunde genommen nur die kodierten Einheiten nacheinander gespeichert werden, da der Dekodierer in der Lage ist, das entsprechende Datenmodell selbst zu erstellen. Gerade bei semiadaptiv arbeitenden Algorithmen, die das Datenmodell an eine endliche Datensequenz anpassen (siehe Def. 5.1.2), ist es jedoch so, dass berechnete Parameter und weitere Information mit übertragen werden müssen, um eine Dekodierung zu ermöglichen.

Adaptivität und Datenzugriff Das moderne Paradigma der Datenkompression ist ein Grundgerüst für adaptive Kompression. Mithilfe der Statistik über bisherige Daten wird das Datenmodell besser an zukünftige Daten angepasst. Im Kontext einer Datenübertragung ist es das Ziel, alle Daten der Reihe nach zu lesen, so dass sich aus der Verwendung von adaptiven Kompressionsmethoden kein Problem ergibt. Im Datenbankenkontext sollen die Daten zwar alle gespeichert, aber nicht zwingend sequentiell gelesen werden. Das bedeutet, dass ein effizienter Datenzugriff gewährleistet sein muss, was sich in den Eigenschaften der verwendeten Algorithmen und Kodierungen und der Datenorganisation niederschlägt. Statt adaptiven oder statischen Methoden werden oft semiadaptive verwendet, denen das moderne Paradigma der Datenkompression nicht genügt.

Das folgende Modell ist ausreichend allgemein, um die verschiedenen Merkmale von Komprimierungsalgoithmen abzubilden und dennoch detailreich genug, um feine Unterschiede zwischen Algorithmen und Mustern von Komprimierungsalgorithmen auszumachen.

2.2 EINFACHES SCHEMA ZUR KOMPRIMIERUNG

Ein allgemeines Schema zur Kompression zeigt Abb. 2.2. Seine Elemente und weitere benötigte Module werden in diesem Kapitel erläutert.

Wortgenerator Der Wortgenerator nimmt sich der Zerlegung der Datensequenz an. Prinzipiell kann man Wortgeneratoren nach ihrer Arbeitsweise in zwei verschiedene Klassen einteilen. Mit einem Datenstrom als Eingabe besteht nur die Möglichkeit, einen endlichen Anfang auszugeben und den Rest ebenso, rekursiv zu verarbeiten

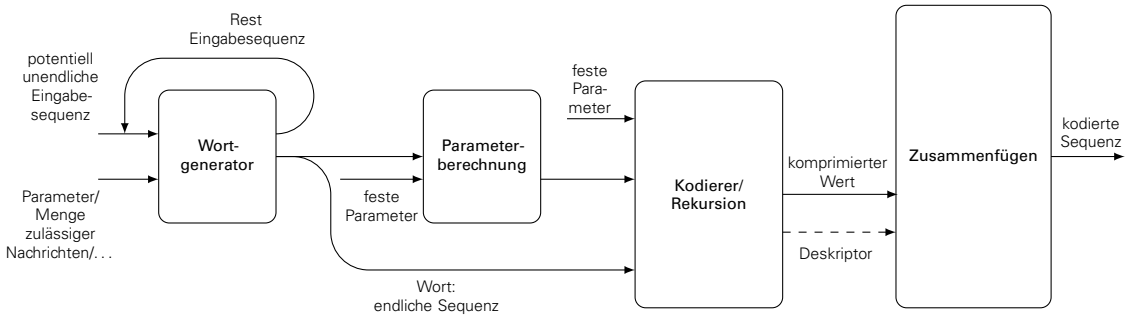
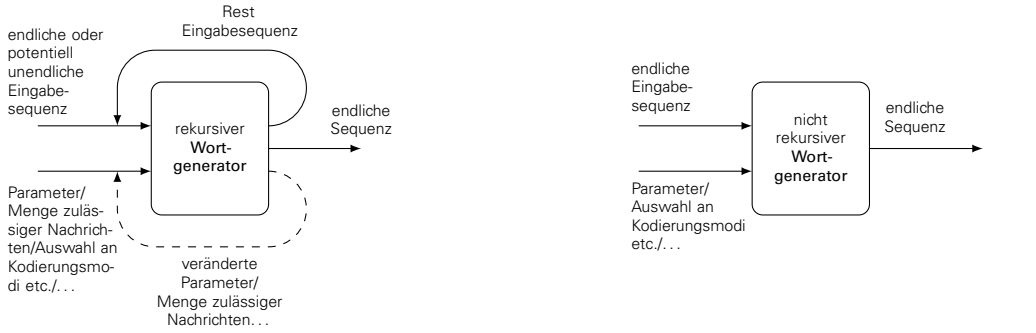


Abbildung 2.2: Allgemeines Kompressionsschema

(siehe Abb. 2.3a). Optionale Datenflüsse werden durch nicht durchgängige Pfeile repräsentiert. Erhält ein Wortgenerator eine endliche Sequenz als Eingabe, so kann die Zerlegungsaufgabe auch nicht rekursiv, zum Beispiel als Optimierungsproblem, gelöst werden (siehe Abb. 2.3b). Hierbei ist oft die Kenntnis der gesamten Eingabe notwendig. Implementierungen von Kompressionsalgorithmen, die Daten vektorisiert verarbeiten, können immer mit einem nicht rekursiven Wortgenerator dargestellt werden. Der Vergleichbarkeit einzelner Algorithmen und Muster halber beginnt jedes Modell für einen Algorithmus mit einem rekursiven Wortgenerator, auch wenn für einen bestimmten Algorithmus, der mit endlichen Sequenzen als Eingabe arbeitet, weder definiert noch wichtig ist, wie seine endliche Eingabe erzeugt wird. Jedes Modularisierungsschema hat eine potentiell unendliche Datensequenz als Eingabe.

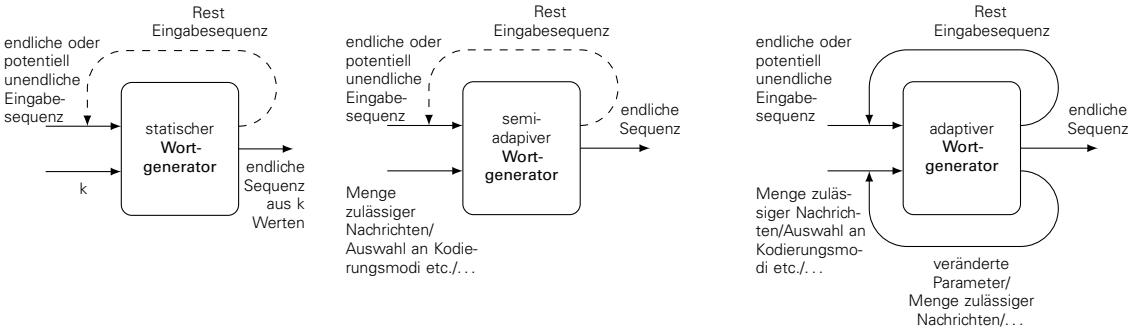
Ein zweites Unterscheidungsmerkmal der Wortgeneratoren ist ihre Adaptivität (siehe Abb. 2.4). Möglich ist eine datenunabhängige (statische) Zerlegung die Eingabe, zum Beispiel werden jeweils 32 Bit oder immer 128 Integerwerte ausgegeben. In diesem Falle wird oft im Anschluss ein Parameter zu berechnen sein, sofern die Eingabe weiter zerlegbare Einheiten aufgeteilt wurde. Die Ausgabe semiadaptiver Wortgeneratoren (z.B. bei RLE) ist nur vom Inhalt der Eingabesequenz abhängig. Adaptive Wortgeneratoren (z.B. bei Relative-10 oder LZ) passen ihr Datenmodell an bisher verarbeitete Daten an, haben also ein „Gedächtnis". Semiadaptive Wortgeneratoren sind ebenso wie adaptive datenabhängig.

Parameterberechnung Semiadaptive, aber auch andere Algorithmen berechnen für endliche Sequenzen Parameter, um die Kodierung den Daten anzupassen. Beispiele sind die Berechnung eines Referenzwertes bei FOR oder die Berechnung der Bitweite beim Binary Packing. Weitere Eingaben für die Berechnung von Parametern sind mögliche Bedingungen oder feste Werte, wie zum Beispiel eine Auswahl an Bitweiten, die man beim Binary Packing zulassen möchte. Prinzipiell kann die Parameterberechnung adaptiv oder semiadaptiv sein (siehe Abb. 2.5). Semiadaptiv berechnete Parameter müssen pro Eingabesequenz als Deskriptor mit gespeichert werden. Bei adaptiver Parameterberechnung wird die Ausgabe als Eingabe für die Berechnung



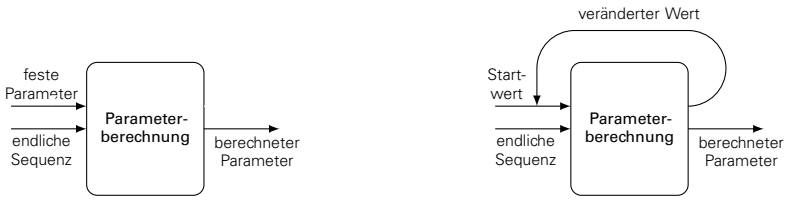
(a) rekursiver Wortgenerator (b) nicht rekursiver Wortgenerator

Abbildung 2.3: Wortgeneratoren mit differierender Verarbeitung der Eingabe



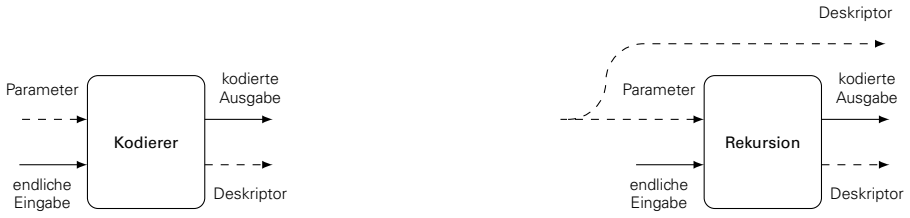
(a) statisch (b) semiadaptiv (c) adaptiv

Abbildung 2.4: Adaptivität von Wortgeneratoren



(a) semiadaptiv (b) adaptiv

Abbildung 2.5: Parameterberechnung



(a) Kodierer (b) Rekursion

Abbildung 2.6: Kodierer/Rekursion

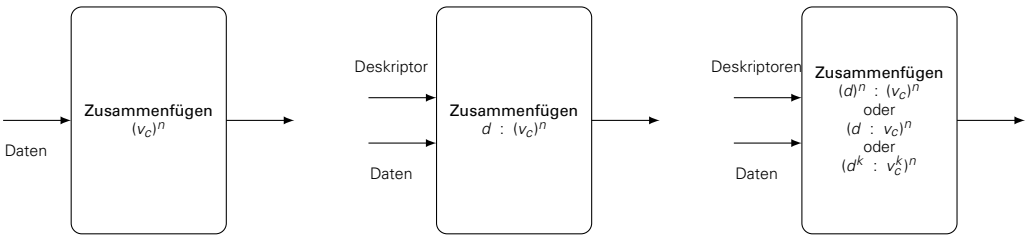
des nächsten Parameters benötigt. Bei einer adaptiven Parameterberechnung genügt die Speicherung eines Startwertes, wenn dieser nicht aus dem Kontext bekannt ist.

In seltenen Fällen benötigt die Parameterberechnung innerhalb einer Rekursion keine endliche Datensequenz als Eingabe, sondern erhält als feste Parameter eine Liste von Werten, die außerhalb der Rekursion berechnet wurden, und deren Länge der Anzahl der zu verarbeitenden Teilsequenzen entspricht. Die Parameterberechnung gibt in diesem Falle pro Teilsequenz einzelne Werte der Liste aus. Ein Beispiel für einen Algorithmus, der dies beinhaltet, ist Simple-16 (siehe Kapitel 4.5.2).

Kodierer/Rekursion Ist eine Sequenz nicht weiter zerlegbar, wird sie kodiert. Ist es möglich, metrische Eigenschaften der Daten zu nutzen, so kann der komprimierte Wert meist einfach berechnet werden, zum Beispiel als eine Differenz zu einem anderen Wert. Gibt es auf Datentypebene keine metrischen Eigenschaften oder sollen diese nicht berücksichtigt werden, wird mithilfe eines Mappings kodiert. Voraussetzung dafür ist, dass für jeden zu kodierenden Wert ein Abbild definiert und die Dekodierbarkeit gegeben ist. Der Kodierer benötigt meistens Parameter, die entweder festgelegt sind oder an irgendeiner Stelle des Algorithmus für eine endliche Sequenz berechnet wurden. Kodierer können auch Deskriptoren berechnen und ausgeben, beispielsweise die Länge von binären Codes oder die Länge eines Laufs bei RLE. Erst beide Ausgaben zusammen besitzen den vollen Informationsgehalt um die Daten zu dekodieren (siehe Abb. 2.6a). Soll eine Sequenz noch weiter zerlegt werden, kann das Komprimierungsschema rekursiv angewendet werden. Dabei können die für die Sequenz berechneten Parameter an anderer Stelle wieder in ein rekursionsinternes Modul wie einen weiteren Kodierer eingehen, wie dies beispielsweise bei der semiadaptiven Kodierung der Fall ist (siehe Abb. 5.2), und als Deskriptor für die gesamte Sequenz dienen (siehe Abb. 2.6b).

Zusammenfügen Die Sequenzen, die der Wortgenerator zerlegt, müssen nach der Kodierung der einzelnen Werte bzw. Sequenzen wieder zusammengefügt werden. Hat ein Wortgenerator einen potentiell unendlichen Datenstrom als Eingabe, so müssen die Daten reihenfolgeerhaltend zusammengefügt werden, da das Modul des Zusammenfügens ebenso einen Datenstrom als Eingabe erhält. Prinzipiell kann man das Modul des Zusammenfügens genauso wie den Wortgenerator als statisch oder datenabhängig klassifizieren. Mit Ausnahme eines einzigen Algorithmus (varint-G8IU) sind aber alle in dieser Arbeit betrachteten Module des Zusammenfügens statisch, weswegen diese Unterscheidung nur von marginalem Interesse ist und der Übersichtlichkeit halber auf eine ebenso komplexe Darstellung wie die des Wortgenerators verzichtet wird.

Oft müssen noch weitere Daten zu einer endlichen Sequenz wie berechnete Parameter als Deskriptoren gespeichert werden. Ohne rekursive Aufrufe kann man die Daten als einfache, potentiell unendliche Sequenz betrachten. Benötigt jeder Daten-



(a) Zusammenfügen mehrerer Werte ohne Deskriptor

(b) Zusammenfügen mehrerer Werte und eines gemeinsamen Deskriptors

(c) Zusammenfügen von Paaren aus Werten und Deskriptoren mit verschiedenen Anordnungen

Abbildung 2.7: Zusammenfügen

satz daraus einen Deskriptor, beispielsweise einen Referenzwert oder eine Bitweite, gibt es die Möglichkeit zunächst den Deskriptor, dann den Datensatz an sich zu speichern oder andersherum. Möglich ist es auch, eine bestimmte Anzahl von Deskriptoren im Wechsel mit der gleichen Anzahl von komprimierten Werten anzuordnen. Bei endlichen Sequenzen, wie sie bei rekursiven Aufrufen entstehen, gibt es noch andere Optionen, die Daten zu organisieren. Beispielsweise ist es denkbar, die Deskriptoren in einem gemeinsamen Bereich und danach alle Datensätze zu speichern. Die verschiedenen Möglichkeiten für statische Module des Zusammenfügens werden gemäß der Notation in Abbildung 2.7 dargestellt, wobei v_c einen kodierten Wert, welcher Ausgabe eines Kodierers oder einer Rekursion war, d einen Deskriptor, der Ausgabe eines Kodierers, einer Rekursion oder einer Parameterberechnung war, $(v_c)^k$, d^k usw. die Konkatenation aus k verschiedenen komprimierten Werten bzw. Deskriptoren bezeichnet. Das Symbol : bezeichnet eine Konkatenation meist verschiedener Arten von Daten, wie Deskriptoren und komprimierten Werten.

2.3 WEITERE BETRACHTUNGEN

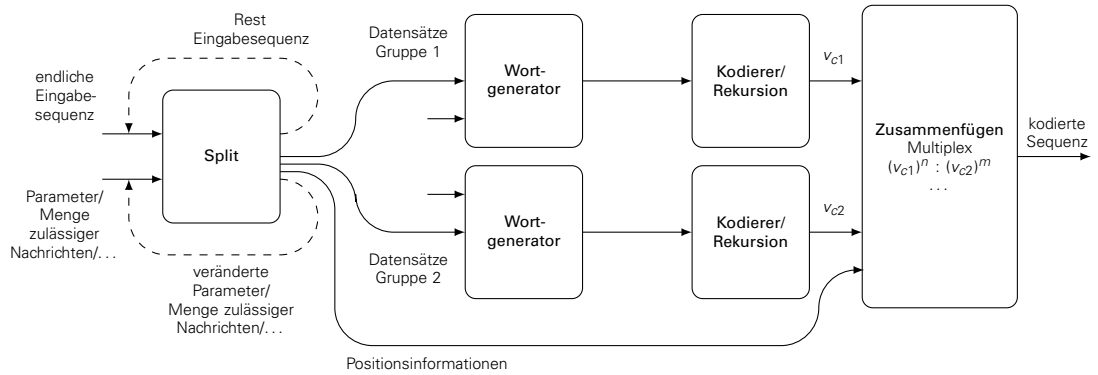
Eine gute Übersichtlichkeit des Kompressionsschemas und damit verbunden eine gute Verständlichkeit sind wünschenswert, doch es bleiben Dinge, die bedacht und betrachtet werden müssen.

2.3.1 SPLITMODUL UND WORTGENERATOR MIT MEHREREN AUSGABEN

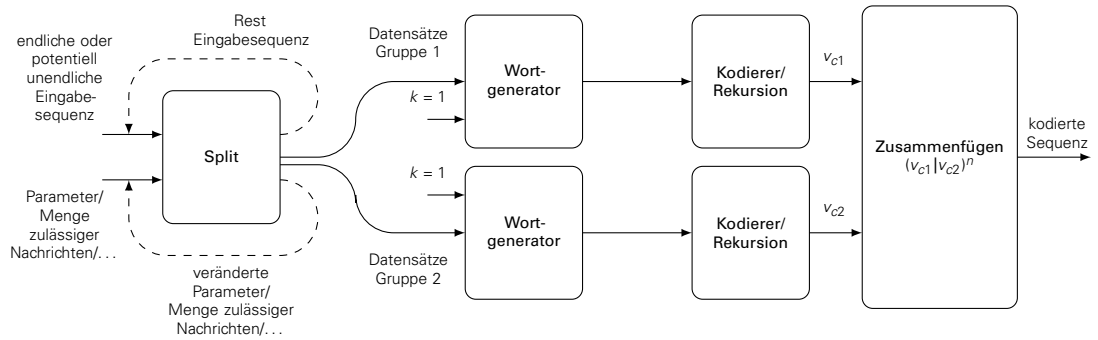
Es gibt Komprimierungsalgorithmen, die die Daten einer Sequenz anhand inhaltlicher Merkmale in mehrere Gruppen unterteilen und diese auf verschiedene Weisen kodieren oder gar umsortieren. Eine solche Aufteilung kann allein mit dem einfachen Schema nicht dargestellt werden. Dafür wird das *Splitmodul* benötigt. Die Eingabe-

sequenz für das Splitmodul muss, wenn es die Werte innerhalb der Sequenz in mehrere Gruppen separiert und die Gruppen nacheinander wieder zusammenfügt, endlich sein (siehe Abb. 2.8a). Anderenfalls, wenn die Daten nur um des Kodierens willen in verschiedene Gruppen aufgeteilt werden und in der ursprünglichen Reihenfolge wieder zusammen gefügt werden, können auch Datenströme als Eingabe dienen. Ein Splitmodul ist ein Wortgenerator, welcher auch nach Adaptivität sowie in die Kategorien rekursiv oder nicht rekursiv eingeordnet werden kann, weswegen die rücklaufenden Datenflüssen optional dargestellt sind. In den in dieser Arbeit betrachteten Algorithmen werden meistens nur einzelne Integerwerte in Gruppen aufgeteilt, sodass kaum ein Algorithmus an dieser Stelle viel Raffinesse beinhaltet. Eine Ausnahme stellt der Algorithmus RLE VByte dar (siehe Kapitel 4.6.6). Jeder der einem Splitmodul nachgeschalteten Wortgeneratoren erhält als Eingabe die Werte einer Datengruppe als Datenstrom, der zum Beispiel aus einzelnen Integerwerten besteht. Die nachgeschalteten Wortgeneratoren werden häufig statisch sein und einzelne Integerwerte ausgeben und in diesem Falle sind sie tatsächlich nicht notwendig. Allerdings besteht auch die Möglichkeit, dass ein nachgeschalteter Wortgenerator den Datenstrom an Werten einer Gruppe nicht in einzelne Werte, sondern statisch oder datenabhängig in mehrere Werte zerlegt. Jedem Wortgenerator folgt ein Kodierer bzw. eine Rekursion. Das entsprechende Kodierer-Modul einer Gruppe g gibt für jeden Eingabewert einen komprimierten Wert v_{cg} aus. Zusammengefügt werden können die Werte der Gruppen nacheinander $((v_{c1})^n : (v_{c2})^m)$, dabei muss die ursprüngliche Reihenfolge der einzelnen Datensätze wieder herstellbar sein, was über die Speicherung von Positionsinformationen als Deskriptoren gelingt. Dieses spezielle Modul des Zusammenfügens wird im Folgenden als *Multiplex* bezeichnet. Die Anzahl der Gruppen ist natürlich nicht auf zwei beschränkt. Die in dieser Arbeit betrachteten Algorithmen mit Splitmodul unterteilen aber nur in zwei Gruppen. Möglich ist, dass die Werte einer Gruppe nicht komprimiert werden, sondern unverändert zusammengefügt werden sollen. Dann ist für diesen Ast weder ein weiterer Wortgenerator noch ein Kodierer nötig. Das Kompressionsschema für PFOR (siehe Abb. 4.7) ist hierfür ein Beispiel. Beispiele für eine Umorganisation der Daten innerhalb eines Algorithmus sind alle Algorithmen des Patched Coding, die in Kapitel 4.4 betrachtet werden.

Eine andere Möglichkeit ist, die Reihenfolge der Daten beizubehalten (siehe Abbildung 2.8b). Hierbei muss die Information, zu welcher Gruppe ein Wert oder eine Sequenz gehört, bewahrt werden. Beim Dekodieren muss bekannt sein, auf welche Art ein Wert kodiert wurde. Das Modul des Zusammenfügens unterscheidet sich von dem, welches bei einer Umorganisation der Daten vonnöten ist, da es die Werte der verschiedenen Gruppen nicht separat zusammenfügt, sondern gemäß der gegebenen Reihenfolge konkateniert. Die Notation $(v_{c1}|v_{c2})^n$ bedeutet, dass komprimierte Werte miteinander konkateniert werden, die entweder mit dem Kodierer für die Daten der Gruppe 1 oder dem Kodierer für die Daten der Gruppe 2 verarbeitet wurden. Da die Werte der verschiedenen Gruppen einzeln nacheinander gespeichert werden, müssen die nachgeschalteten Wortgeneratoren statisch einzelne Werte ($k = 1$) aus-



(a) Separation der Gruppen bei endlicher Eingabesequenz



(b) Beibehaltung der Reihenfolge

Abbildung 2.8: Getrennte Kodierung zweier Datengruppen

geben, damit einzelne Werte kodiert werden. Alles andere wäre an sich nicht plausibel, weil nicht vorhersagbar ist, zu welcher Gruppe der nachfolgende Wert gehört und es wenig Sinn ergibt, mehrere Werte, die möglicherweise nicht aufeinanderfolgend gespeichert werden sollen, gemeinsam zu kodieren.

Einige Algorithmen aus der PFOR-Familie (Kapitel 4.4), die Integerwerte komprimieren, unterteilen endliche Sequenzen von mit 32 Bit kodierten Werten in höhere und niedrigere Bits. Die Sequenz aller höheren Bits wird anders kodiert als die Sequenz mit den niedrigeren Bits. Aus Gründen der Übersichtlichkeit werden Wortgeneratoren mit mehreren Ausgängen zugelassen, wenn die Ausgabewerte nicht aufgrund inhaltlicher Merkmal wie beim Splitmodul auf mehrere Gruppen aufgeteilt werden, sondern aufgrund positionaler Eigenschaften, so dass eine semantische Einheit (hier ein Integerwert) zerlegt wird. Ein Beispiel hierfür zeigt Abbildung 2.9. Eingabe ist eine Sequenz aus mit 8 Bit kodierten Werten. Jeweils einen Ausgang gibt es für die fünf höheren Bits $b_{\uparrow}$ und die drei niedrigeren Bits $b_{\downarrow}$.

2.3.2 HIERARCHISCHE DATENORGANISATION

Eine gute visuelle Übersicht über die Organisation von Werten, Sequenzen und zugeordneten Deskriptoren erhält man oft durch eine baumartige Darstellung. Die Bausteine Wortgenerator und Zusammenfügen sind als komplementäre Module gedacht.

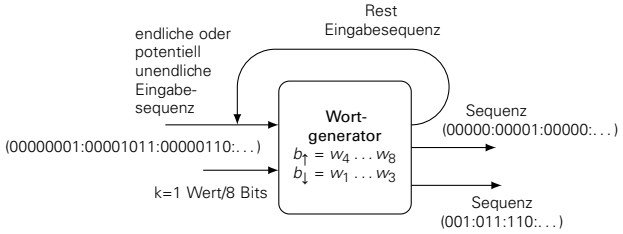


Abbildung 2.9: Wortgenerator mit zwei Ausgaben

Alle Sequenzen, die zerlegt werden, müssen am Ende wieder zusammengefügt werden.

Wird das einfache Kompressionsschema aus Abb. 2.2 ohne Rekursion ausgeführt, wird der Eingabedatenstrom unterteilt, kodiert und wieder zusammengefügt. Möglicherweise wird zu jedem Wert ein Deskriptor berechnet. Diese einfache Möglichkeit der Datenorganisation zeigt Abb. 2.10a. Mögliche Deskriptoren sind als Kreise dargestellt, Werte, die durch den Wortgenerator erzeugt wurden, durch Quadrate an den Blattknoten des Baumes. Die Wurzel bezeichnet den gesamten Eingabedatenstrom. Zeichnet sich das Schema für einen Algorithmus durch eine Rekursion aus, wird jede erzeugte endliche Sequenz ein weiteres Mal durch einen zweiten Wortgenerator zerlegt und nach der Kompression der einzelnen Einheiten wieder zusammengefügt.

Außerdem wird möglicherweise für jede endliche Sequenz ein Parameter berechnet, der als gemeinsamer Deskriptor für mehrere Werte dienen kann (siehe Abbildung 2.10b). Die Daten sind hier mehrstufig organisiert. Die Parameter für endliche Sequenzen sind als Kreise an inneren Knoten sichtbar.

Abbildung 2.10c zeigt die Datenhierarchie für ein Schema mit einer Rekursion. Innerhalb der Rekursion gibt es ein Splitmodul, welches die Daten aufgrund inhaltlicher Merkmale aufteilt, damit diese mit verschiedenen Kodieren komprimiert werden. Ein Splitmodul bewirkt, dass unterschiedliche Daten auf verschiedene Weise kodiert werden können, möglicherweise auch mit verschiedenen Rekursionstiefen, die Knoten auf einer Ebene können sich in ihrer Struktur unterscheiden. Im Beispiel berechnet einer der beiden Kodierer zu jedem komprimierten Wert noch einen Deskriptor, dargestellt als Kreis und Rechteck, der andere berechnet nur einen komprimierten Wert ohne Deskriptor.

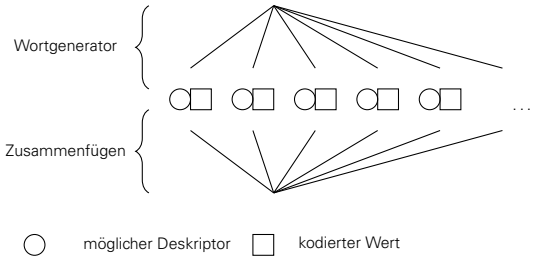
Im Zusammenspiel mit einem Multiplexmodul gelingt eine Umordnung der kodierten Daten (siehe Abb. 2.10d). Die dargestellte Datenhierarchie beruht auf der Anwendung des einfachen Kompressionsschemas mit Rekursion. Innerhalb der Rekursion existiert ein Splitmodul, welche die Daten in zwei Gruppen mit zwei verschiedenen Kodierern aufteilt. Einer der beiden Kodierer berechnet zu den zu komprimierenden Werten seiner Gruppe keinen Deskriptor, der andere komprimiert die Werte seiner Gruppe und berechnet als Deskriptor eine Positionsangabe. Deshalb können die Wer-

te beider Gruppen getrennt und physisch an verschiedenen Stellen gespeichert werden, dargstellt durch einen Trennstrich. Abbildung 2.10d zeigt ein erläuterndes Beispiel. Der erste Wortgenerator gibt die endliche Sequenz (3:5:7:2:3) aus. Das Splitmodul unterteilt die Werte in solche, die mit zwei Bit binär kodiert werden können (3, 2 und 3), und solche, die mit mehr als zwei Bit kodiert werden müssen (5 und 7). Der Deskriptor am inneren Knoten, der für die gesamte Sequenz berechnet wurde, enthält die Bitweite $bw = 2$ sowie die Information, an welcher Position der erste Wert der zweiten Gruppe steht ($pos = 2$). Die nachfolgenden Deskriptoren entsprechen dem Abstand zum nächsten Wert aus der zweiten Gruppe (1 und ein definierter Abbruchwert 0 mit der Bedeutung, dass keine weiteren Ausnahmen folgen). So ist aus einer kodierten Sequenz die ursprüngliche Reihenfolge der Werte wiederherstellbar.

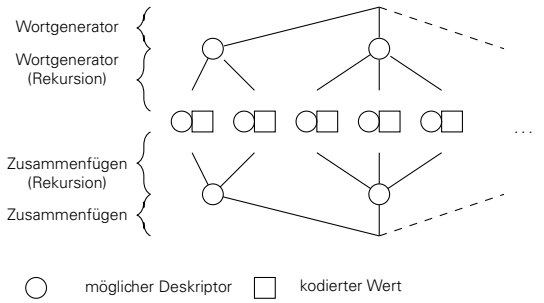
Wortgeneratoren nach Splitmodulen eröffnen keine neue Ebene in der Datenhierarchie. Auch der Wortgenerator mit mehreren Ausgaben ist so gedacht, dass jede Ausgabe als Datenstrom in einen weiteren, normalen Wortgenerator eingehen kann, ohne dass eine neue Ebene eröffnet wird. Schaltet man zum Beispiel in Abb. 2.9 dem zweiten Ausgang für die niedrigeren Bits einen Wortgenerator nach, zerlegt dieser nicht etwa endliche Sequenzen, die aus 3 Bits bestehen, in einzelne Bits, sondern die Sequenz von aus 3 Bits bestehenden Werten in endliche Sequenzen von 3-Bit-Werten bzw. einzelne 3-Bit-Werte. Eine weitere Ebene ließe sich bei Bedarf durch den Anschluss einer Rekursion eröffnen. Die Module werden in diesem Sinne verwendet, sorgen allerdings für eine gewisse Veruneinheitlichung, was die hierarchische Datenorganisation betrifft.

2.3.3 MEHRMALIGER AUFRUF DES SCHEMAS

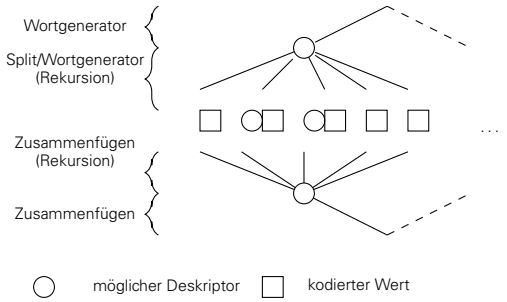
Es kann vorkommen, dass der mehrmalige Aufruf des gesamten Schemas nötig ist, wenn eine Sequenz mit zwei Kompressionsalgorithmen verarbeitet wird. Dies ist nötig, wenn die hierarchische Datenorganisation bei der ersten Verarbeitung nicht konsistent zu der bei der zweiten Verarbeitung ist. Anderenfalls ist eine Rekursion nutzbar. Beispielsweise soll im ersten Durchlauf eine Differenzkodierung vorgenommen werden. Dabei kann der Datenstrom nicht unterteilt werden, da die Kodierung jedes Wertes von allen Vorgängerwerten abhängt. Sollen im zweiten Durchlauf zum Beispiel immer 128 Werte betrachtet und mit gemeinsamer Bitweite kodiert werden, so gelingt keine Darstellung mit rekursiven Aufrufen des Kompressionsschemas. Es gibt nur die Möglichkeit, das Kompressionsschema zweimal nacheinander aufzurufen.



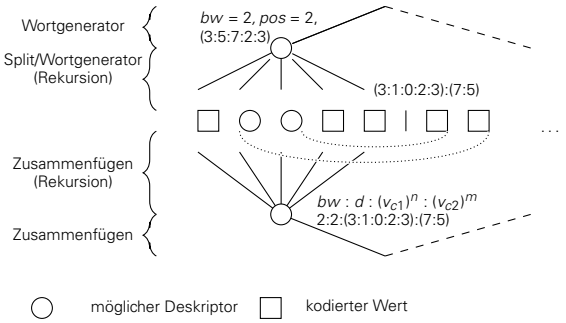
(a) Einfache Modularisierung ohne Rekursion



(b) Modularisierung mit Rekursion



(c) Differierende Kodierung auf der gleichen Ebene durch Splitmodul



(d) Umorganisation der Daten mit Split und Multiplex-Modul

Abbildung 2.10: Gestaltung der Datenhierarchie durch Einsatz verschiedener Module: Wortgeneratoren, Parameterberechnung, Split und Zusammenfügen

2.4 BEWERTUNG UND BEGRÜNDUNG DER MODULARISIERUNG

Eine gute Modularisierung zeichnet sich durch verschiedene Eigenschaften aus. Die für diese Abstraktionsebene relevanten Merkmale sind dabei folgende.

Hohe Modulbindung (Kohäsion). Die Kohäsion ist der Grad des Zusammenhangs zwischen den einzelnen Daten und Operationen desselben Moduls. Jedes Modul außer einer Rekursion realisiert eine im Normalfall nicht sinnvoll weiter unterteilbare Operation. Eine ausnahme stellt die Parameterberechnung dar. Dieses Modul wird der Übersichtlichkeit halber pro Ebene nur einmal dargstellt, obwohl es möglich ist, dass mehrere unabhängige Parameter berechnet werden.

Niedrige Modulkopplung. Die Modulkopplung ist der Grad des Zusammenhangs zwischen verschiedenen Modulen. Dieser unterscheidet sich je nach Algorithmus recht stark. Teilweise ergeben sich Parameter der Parameterberechnung bereits im Modul des Wortgenerators. Bei manchen Algorithmen ist sogar die Trennung zwischen Wortgenerator und Kodierer nicht notwendig. Um den Ausprägungen aller Algorithmen gerecht zu werden, ist die gegebene Trennung der Module aber durchaus sinnvoll.

Minimale Schnittstelle. Je kleiner die Schnittstelle, desto geringer ist die Gefahr einer hohen Modulkopplung. Deswegen werden möglichst wenige Eingaben benötigt und möglichst wenige Ausgaben erzeugt.

Hohe Interferenzfreiheit. Nebenwirkungen auf andere Module sind zu vermeiden, da nicht erwünscht, weil diese die Änderbarkeit und Erweiterbarkeit einzelner Module negativ beeinflussen. Module sind oft leicht ersetzbar durch andere mit der gleichen Schnittstelle.

2.5 ZUSAMMENFASSUNG

In der Literatur vorhandene Modularisierungsideen für Datenkompression im Allgemeinen sind gerade für Algorithmen, die endliche Sequenzen semiadaptiv komprimieren, nicht ausreichend. Die hier eingeführte Modularisierung genügt den Anforderungen Sequenzen überhaupt und auch mehrstufig und adaptiv zerlegen zu können, die für die Kodierung nötigen Parameter für endliche Sequenzen berechnen zu können und die Reihenfolge der Daten umzugestalten. Die Güte sowie der Sinn und Nutzen der allgemeinen Modularisierung wird in den den folgenden Kapiteln anhand

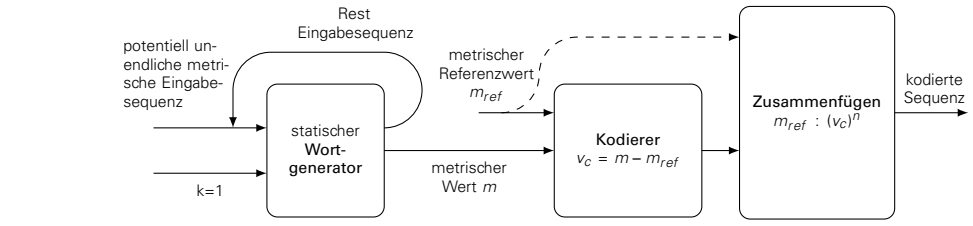
von Mustern, konkreten Algorithmen und Modellierbarkeit von Kompressionseigenschaften auf die Probe gestellt.

3 MODULARISIERUNG FÜR VERSCHIEDENE KOMPRESSIONSMUSTER

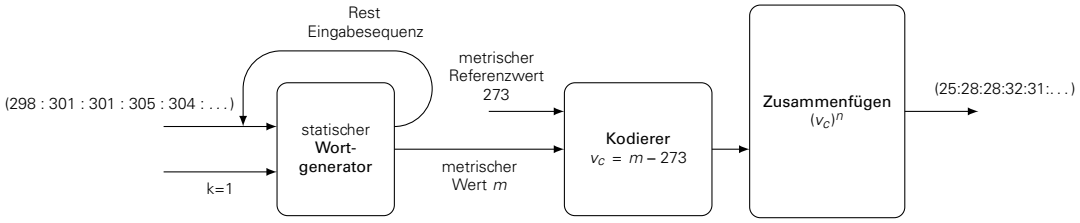
Dieses Kapitel dient dazu, sechs allgemeinen Techniken, *Frame of Reference* (FOR), *Differenzkodierung* (DELTA), *Symbolunterdrückung*, *Lauflängenkodierung* (RLE), Wörterbuchkompression (DICT) sowie *Bitvektoren* (BV) vorzustellen, zu modularisieren und zu vergleichen. Mindestens eine dieser Techniken ist in jedem Kompressionsalgorithmus enthalten. Es gibt den Gedanken, dass die sechs Techniken Kategorien bzw. Mengen darstellen, die sich so in Beziehung bringen lassen, dass man sie im Idealfall über- oder unterordnen kann. Aus diesem Blickwinkel betrachtet ist jede Technik eine Wörterbuchkompression. Es existieren Schnittbereiche, z.B. zwischen Lauflängenkodierung und Symbolunterdrückung oder Frame of Reference und Differenzkodierung. Bitvektoren, als 1-aus-*n*-Code betrachtet, sind ein Wörterbuchalgorithmus. Modularisiert man die sechs Techniken mit dem eingeführten Kompressionsschema, so fallen vielmehr Analogien zu Entwurfsmustern in der Softwareentwicklung auf. Auch hier stehen die verschiedenen Module in bestimmten Weisen miteinander in Beziehung und zeichnen sich durch mehr oder weniger speziell definierte Inhalte aus, was im Folgenden betrachtet werden soll.

3.1 FRAME OF REFERENCE (FOR)

Für die allgemeine Definition dieser Methode muss die Eingabesequenz aus metrischen Werten bestehen, wie zum Beispiel rationalen Zahlen oder Tupeln aus rationalen Zahlen, auf denen eine Addition definiert ist. Alle Werte werden als Differenz zu einem gemeinsamen Referenzwert m_{ref} kodiert. Bei sinnvoller Wahl desselben können damit kleinere Werte kodiert werden. Dieser ganz allgemeinen Auffassung genügt das statische Schema in Abb. 3.1. Hier ist der Referenzwert aus dem Kontext



(a) Kompressionsschema



(b) Beispiel

Abbildung 3.1: Allgemeines FOR-Schema

bekannt. Der Wortgenerator gibt vom Anfang der Sequenz jeweils einen Integerwert aus. Dieser wird dann als Differenz zum Referenzwert kodiert. Das Modul des Zusammenfügens konkateniert den Referenzwert m_{ref} mit allen kodierten Werten und gibt dies als kodierte Sequenz $m_{ref} : (v_c)^n$ aus. Notwendig ist die Speicherung des Referenzwertes nur, wenn dessen Kenntnis für das Dekodieren nicht vorausgesetzt werden kann. Dies ist durch einen optionalen Pfeil dargestellt. Ein Beispiel ist die kontinuierliche Messung von absoluten Temperaturwerten einer Wasserkühlung, die in $°C$ gespeichert werden sollen, da aus dem Kontext bekannt ist, dass größere Messwerte als $273K$ zu erwarten sind. Dabei wird der Referenzwert $m_{ref} = 273$ gesetzt. Dieser muss in diesem Falle nicht gespeichert werden, wenn die genutzte Temperaturskala bekannt ist. Abbildung 3.1b zeigt hierzu ein Beispiel. Meist gehört es zum Selbstverständnis, dass der Referenzwert für eine endliche Sequenz wie in Abb. 3.2a aus den gegebenen Daten berechnet und als Deskriptor gespeichert wird. Notwendig ist dies jedoch nicht. In dieser und den folgenden Abbildungen sind nur die schwarz dargestellten Module und Datenflüsse relevant und so in der Form zwingend. Ein konkreter Algorithmus muss nur das Zusammenspiel aus schwarz dargestellten Modulen und Datenflüssen aufweisen, einem vorgestellten Muster zu Genüge zu tun. Grau dargestellte Module zeigen nur einen möglichen allgemeinen Kontext. Im Beispiel in Abb. 3.2b werden immer vier Werte mit einem gemeinsamen Referenzwert kodiert. Der erste Wortgenerator erhält die Anweisung, immer vier Werte auszugeben. Der von der Parameterberechnung ausgegebene Referenzwert m_{ref} ist der kleinste der vier Werte. Angenommen wird, dass der kleinste Wert einer endlichen Sequenz ein sinnvoller Referenzwert ist und dieser sich von Zeit zu Zeit, nicht öfter als aller vier Werte ändert. Die Sequenz aus vier Werten sowie der berechnete Referenzwert gehen in die Rekursion ein. Der zweite Wortgenerator gibt einzelne Zahlen aus. Von jeder der 4 Zahlen wird der gemeinsame Referenzwert subtrahiert. Anschließend

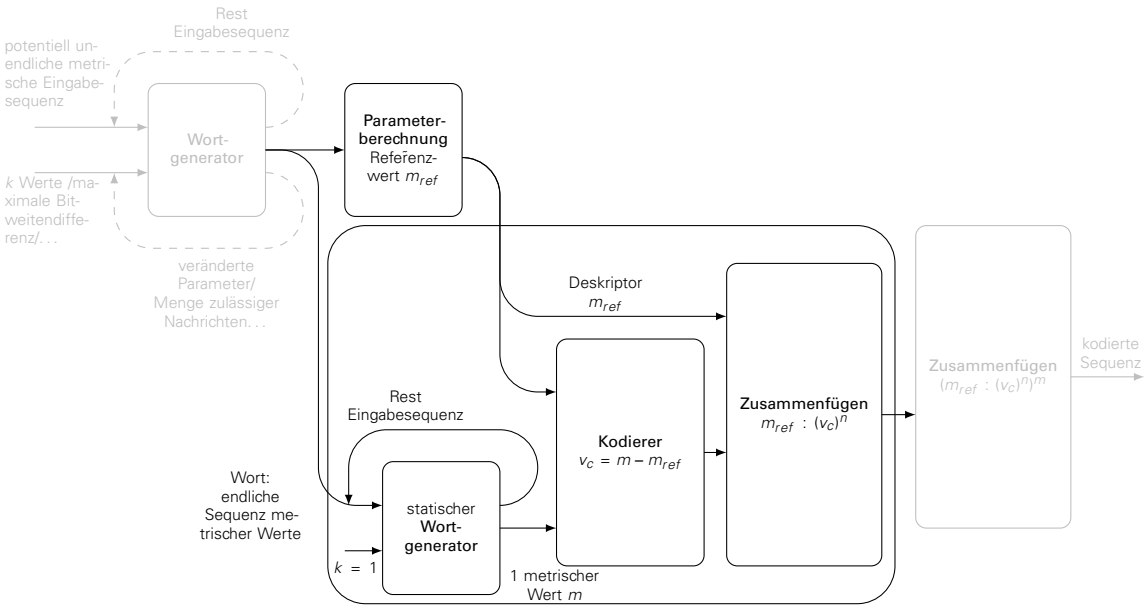
werden im ersten Modul des Zuammenfügens der Referenzwert und die vier komprimierten Werte konkateniert. Gewünscht ist, dass trotz des Overheads weniger Speicherplatz benötigt wird. Das äußere Modul des Zusammenfügens konkateniert alle Datenbereiche aus jeweils vier Werten.

Für viele Autoren gehört neben dem Referenzwert als untere Grenze auch die (durch die Bitweite festgelegte) obere Grenze direkt zu FOR, während in dieser Arbeit das FOR-Muster an sich nur auf abstrakter Ebene der natürlichen Zahlen mit dem Referenzwert als gemeinsamen Deskriptor für mehrere Werte betrachtet wird.

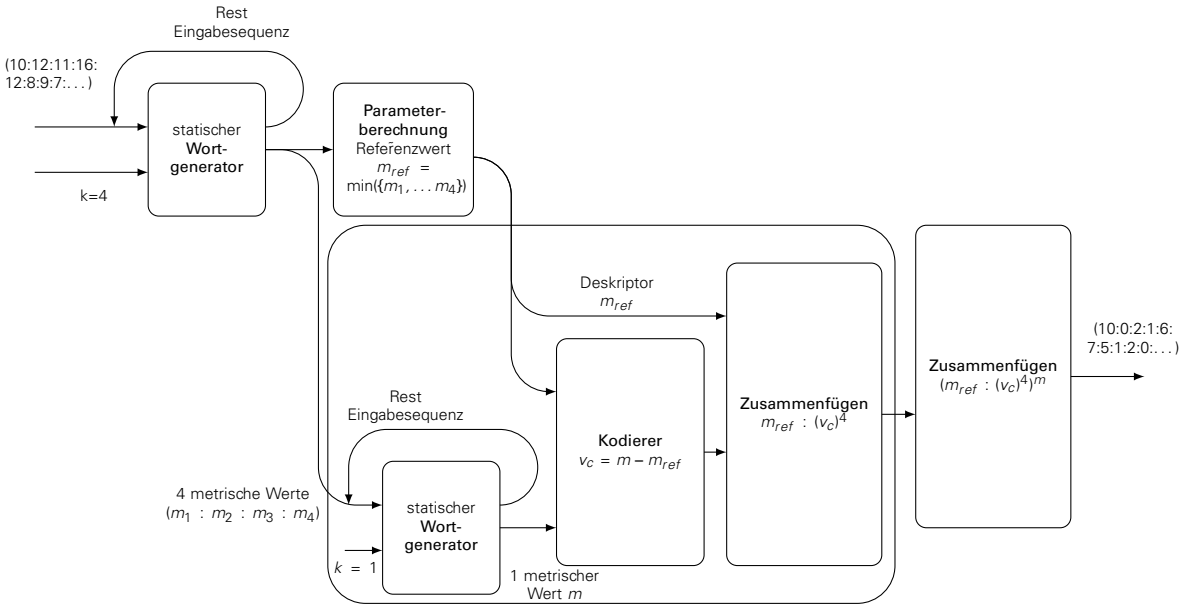
3.2 DIFFERENZKODIERUNG (DELTA)

Auch Differenzkodierungsverfahren können als Eingabe nur metrische Werte akzeptieren. Sie werden unter der Annahme angewendet, dass die Differenzen zwischen einzelnen Werten kleiner sind als die absoluten Originalwerte. Bei sortiert vorliegenden natürlichen Zahlen ist das immer der Fall. In Kombination mit Kodierungen, bei denen kleinere Zahlen durch kürzere Codes repräsentiert werden können, benötigen kleinere Werte geringere Bitweiten. Der Wortgenerator gibt, genau wie der beim FOR, einzelne Zahlen m aus. Differenzverfahren haben durch die Datenabhängigkeit vom Vorgängerwert per se adaptive Anteile (siehe Abb. 3.3). Das Modul der Parameterberechnung erhält neben der Ausgabe m des Wortgenerators einen Parameter p als Eingabe. Der Wert p ist zu Beginn mit $p = 0$ initialisiert. Die Eingabe m der Parameterberechnung dient im nächsten Schritt wiederum als Eingabe p für den nächsten Wert. Die größte Gemeinsamkeit mit FOR besteht darin, dass beide Verfahren exakt den gleichen Kodierer nutzen. Beide Verfahren unterscheiden sich jedoch in der Art und Weise, wie die Daten zusammengefügt werden. Die Differenzkodierung benötigt keinen Deskriptor, weil in Normalfall der Startwert für die Parameterberechnung mit $p = 0$ initialisiert ist und dieser Wert nicht explizit erwähnt werden muss.

Die Beispieldaten in Abbildung 3.3 werden wie folgt im Schema verarbeitet. Der Wortgenerator gibt den ersten Wert der Eingabesequenz $m = -2$ aus. Der Rest der Sequenz wird für die Komprimierung der nächsten Zahl erneut eingelesen. Der Wert $m = -2$ ist ebenso wie der Startwert $p = 0$ Eingabe für die Parameterberechnung. Die Parameterberechnung gibt als Referenzwert $m_{ref} = 0$ aus sowie den ursprünglichen Eingabewert als $p = -2$, der für die Berechnung des Referenzwertwertes des nächsten Wertes benötigt wird und als Eingabe dient. Der Kodierer gibt die Differenz $v_c = (-2)-0 = -2$ aus. Als zweiten Wert gibt der Wortgenerator den Wert $m = -1$ aus, welcher gemeinsam mit $p = -2$ in die Parameterberechnung eingeht, die wiederum $m_{ref} = -2$ sowie $p = -1$ ausgibt. Der Kodierer kodiert $m = -1$ mit $v_c = (-1) - (-2) = 1$. Das Modul des Zusammenfügens konkateniert alle Werte ohne Deskriptor.



(a) Kompressionsschema



(b) Beispiel

Abbildung 3.2: Semiadaptives FOR-Schema mit berechnetem Referenzwert

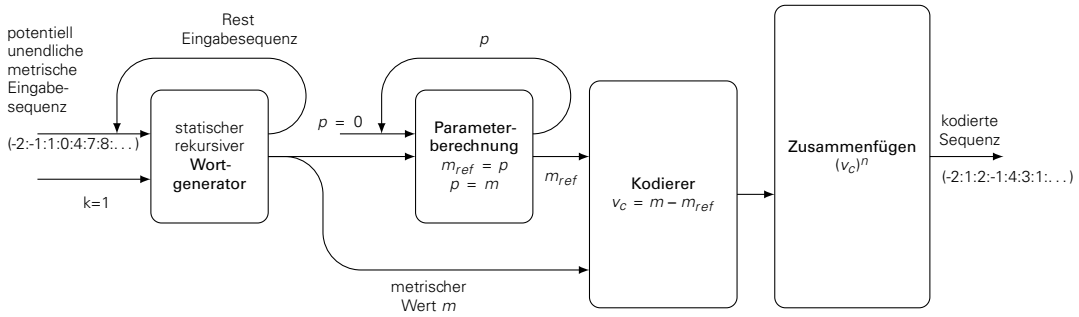


Abbildung 3.3: Differenzverfahren: Kompressionsschema mit Beispiel

3.3 SYMBOLUNTERDRÜCKUNG

Unter dem Begriff *Nullenunterdrückung* werden sehr verschiedene Komprimierungsverfahren zusammengefasst. [Aro77] widmet sich unter diesem Thema den Präsenzbits sowie der Lauflängenkodierung für endliche oder potentiell unendliche Sequenzen, in denen Nullen auftauchen. [RH93] versteht darunter eine Lauflängenkodierung von Nullen und Leerzeichen. Darunter fiele auch die Unterdrückung von Leerzeichen im Wörterbuchalgorithmus ALM [ALM96]. Neben dem Begriff *Null suppression* gibt es in der englischsprachigen Literatur noch den Begriff *Zero suppression*, der eher für die Eliminierung führender und damit redundanter Nullen bei binär kodierten Zahlen gebräuchlich ist. Alle dieser Methoden haben gemeinsam, dass es im Zeichenvorrat ein ausgezeichnetes Symbol *s* gibt, welches sich meist semantisch von allen anderen abhebt, weswegen sich der Begriff *Symbolunterdückung* für eine Zusammenfassung dieser Methoden besser eignet. Das ausgezeichnete Symbol wird im Wortgenerator oder im Kodierer anders behandelt als andere Symbole. Im Falle von Präsenzbits ist dieses Symbol der NULL-Wert. Nullen sind das neutrale Element der Addition. Die genaue Anzahl führender Nullen beeinflusst Additionsoperationen nicht und ist damit an sich schon eine redundante Information. Leerzeichen dienen in allen Sprachen dazu, Wörter voneinander zu separieren. Die Anzahl von Wiederholungen von Leerzeichen zwischen konkatenierten Wörtern besitzt auf semantischer Ebene keinerlei Bedeutung. Viele der Algorithmen, aber nicht alle, sind RLE-Komprimierungen. Für Symbolunterdrückungsmuster lässt sich allgemein keine Modularisierung darstellen, da es sich einfach nur durch die Sonderbehandlung eines Symbols auszeichnet. In Kombination mit einer Lauflängenkodierung lässt es sich aber mit dem Kompressionsschema ausdrücken.

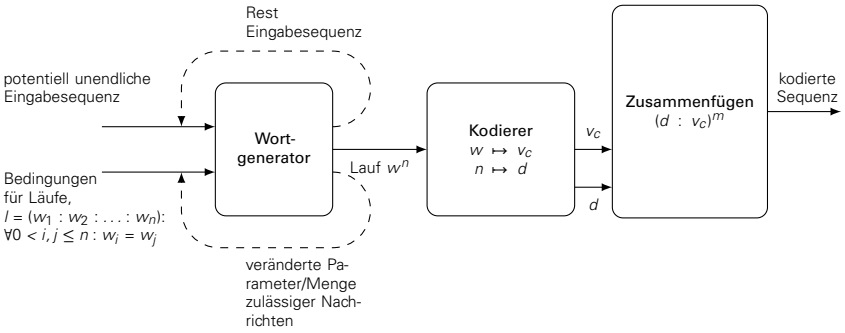
3.4 LAUFLÄNGENKODIERUNG (RLE)

Merkmale dieses Musters sind das Vorhandensein von Läufen w^n, endlichen Sequenzen aus ein und demselben Wert. Werden wirklich einfach nur Läufe von Werten kodiert, so reicht das simple Kompressionsschema in Abbildung 3.4a aus. Der Wort-

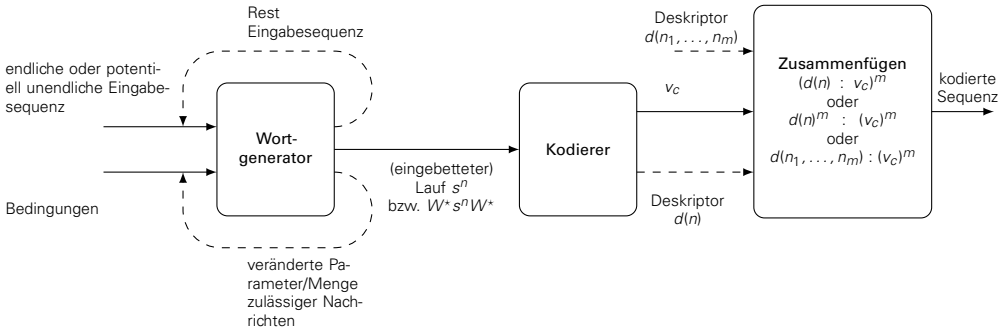
generator unterteilt den Eingabedatenstrom in Läufe. Im Kodierer werden dann der Wert w an sich und die Lauflänge n kodiert. Läufe können auch in einer Sequenz zum Beispiel als führende Nullen eingebettet sein. Solche Fälle liegen im Schnittbereich zwischen Symbolunterdrückung und Lauflängenkodierung. Dies ist in Abbildung 3.4b dargestellt. Die Informationen über Sequenzlängen werden beim Zusammenfügen vor allem in der Form $(d(n) : v_c)^m$ oder $d(n)^m : (v_c)^m$ bei mehreren Läufen gleicher Länge als Deskriptor gespeichert. Dabei ist $d(n)$ eine eineindeutige Abbildung, oft die Identitätsabbildung. Werden mit 32 Bits kodierte Werte mit weniger Bits gespeichert und die Bitweite als Deskriptor angegeben, so die de Bitweite $bw = d(n)$ eine Funktion von n, der Anzahl der führenden Nullen, die entfernt wurden. Jeder Wert w einer endlichen Sequenz hat eine komprimierte Form v_c und eine Lauflänge n. Beide werden entweder zusammen oder in der Gruppe aus Deskriptoren und einer Gruppe aus komprimierten Werten gespeichert. Für verschiedene Algorithmen aus der Simple-Familie (siehe Abschnitt 4.5) gibt es auch eine Form $d(n_1, \ldots, n_m) : (v_c)^m$, wobei $d(n_1, \ldots, n_m)$ den Kodierungsmodus bezeichnet. Besitzen mehrere Werte einen gemeinsamen Deskriptor, dann wurde letzterer vor der Generierung der Läufe festgelegt. Abbildung 3.4c zeigt ein Beispiel für eine einfache Symbolunterdrückung mit Lauflängenkodierung. Der Wortgenerator gibt mit 8 Bits kodierte Integerwerte > 0 aus, der Kodierer komprimiert jeden Wert, indem er führende Nullen entfernt. Diesen komprimierten Wert sowie die Bitweite als Deskriptor gibt er aus. Das Modul des Zusammenfügens konkateniert alternierend Bitweiten und komprimierte Werte.

3.5 WÖRTERBUCHKOMPRESSION (DICT)

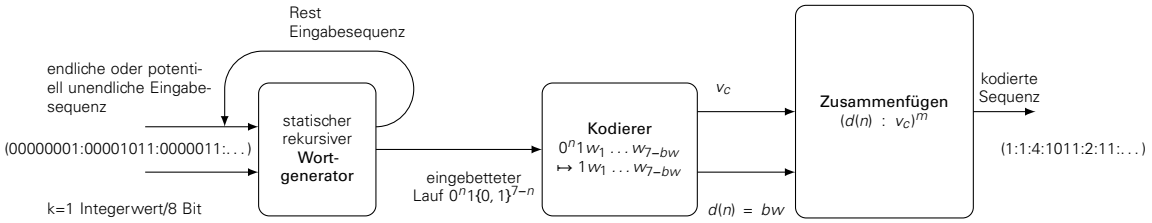
Wörterbuchverfahren sind Kompressionsmethoden, die jeden Wert w aus einem Wertevorrat W eindeutig auf ein Codewort v_c aus einem Code C abbilden. Möglich ist bei einer gegebenen Kodierung aus W' nach C auch eine Wörterbuchabbildung aus W nach W'. Ein Wörterbuchalgorithmus muss nicht zwangsläufig auf einen binären Code abbilden, möglich ist auch eine Abbildung auf abstrakterer Ebene. Beispielsweise ist FOR eine Wörterbuchtechnik, die von der Menge $\mathbb{N}$ in die Menge $\mathbb{N}$ abbildet. Diese Zuordnung muss bei verlustfreier Kompression eineindeutig sein. Abbildung 3.5 zeigt das allgemeinste Schema für Wörterbuchverfahren. Wie jedes Kompressionsschema benötigt ein solches Verfahren einen Wortgenerator zu Beginn und am Ende ein Modul des Zusammenfügens. Worauf es ankommt, ist allerdings das Vorhandensein eines Kodierers, der jeden möglichen Eingabewert aus einer Menge von Werten in eine Menge von Werten abbildet, also jeden Wert ersetzt und ausgibt. Dieses triviale Muster findet sich in jedem Kompressionsschema wieder, da jedes Kompressionsschema einen Kodierer benötigt. Abbildung 3.5 zeigt nur die einfachste Möglichkeit eines Wörterbuches. Natürlich können auch Parameter berechnet werden und Rekursionen im Schema vorhanden sein, so wie es bei FOR der Fall ist.



(a) Einfache Lauflängenkodierung



(b) Symbolunterdrückung mit RLE-Schema



(c) Beispiel: Symbolunterdrückung mit RLE-Schema

Abbildung 3.4: Lauflängenkodierung

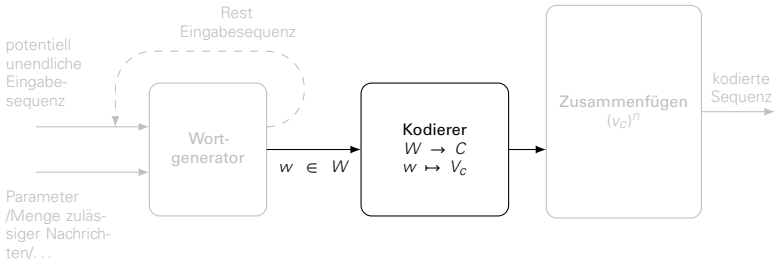


Abbildung 3.5: Wörterbuchkompression

Ein Beispiel für einen einfachen Wörterbuchalgorithmus, der Werte nicht berechnet, sondern die Abbildung kennt, ist die Kodierung von Unicodezeichen mit UTF-8.

3.6 BITVEKTOREN (BV)

Bitvektoren sind ein Wörterbuchverfahren, bei dem jedem Wert ein Schlüsselwert aus einem 1-aus-*n*-Code zugeordnet wird. Diese Auffassung ergibt auf praktischer Ebene nicht allzu viel Sinn, aber für das Modulariserungsschema ist sie durchaus nutzbar. Bedingung für diese Verfahren ist, dass der Wertevorrat zu Beginn feststeht. Bei statischen Bitvektorverfahren ist der Wertevorrat aus dem Kontext bekannt, bei semiadaptiven Verfahren wird er im ersten Pass bestimmt (siehe Abb. 3.6). Die Zerlegung in eine endliche Sequenz wird als gegeben vorausgesetzt und ist bei dieser Technik nicht von Bedeutung. Im Beispiel in der Abbildung soll die aus $l = 6$ Werten bestehende Sequenz (*DE*:*DE*:*FR*:*DE*:*EN*:*EN*) mit Bitvektoren komprimiert werden. Dazu wird zunächst das Wörterbuch berechnet. Bei drei verschiedenen Werten ergibt sich $n = 3$. Die Parameterberechnung erstellt ein Wörterbuch, welches die verschiedenen vorkommenden Werte auf einen 1-aus-3-Code abbildet. Es enthält die Tupel (*DE,* 100), (*FR,* 010) und (*EN,* 001). Der Kodierer kennt das berechnete Wörterbuch und komprimiert jeden Wert mit dieser Abbildung, beispielsweise *EN* mit 001. Das Modul des Zusammenfügens fügt alle Informationen zusammen. Dazu gehört das Wörterbuch. Möglich ist, seine Größe *n* und danach die Konkatenation aller Einträge nach der Ordnung der zugeordneten Codewörter zu speichern. Für die drei verschiedenen Werte ist jeweils ein Bitvektor der Länge $l = 6$ definiert, der angibt, ob an einer Position in der Sequenz dieser Wert vorkommt oder nicht. Der Vektor für den Wert *EN* ergibt sich zu 000011, was der Konkatenation des 3. Bits aller komprimierten Werte der endlichen Sequenz entspricht. Beim Zusammenfügen wird das Wörterbuch mit allen $n = 3$ Bitvektoren konkateniert.

3.7 VERGLEICH VERSCHIEDENER MUSTER UND TECHNIKEN

So verschieden die vorliegende Information über zu komprimierende Daten, Verteilungen, Voraussetzungen, Wertebereiche, Redundanzarten etc. sein kann, so verschieden sind auch die gedanklichen Ansätze, die genutzt werden, um Daten (verlustfrei) zu komprimieren. Eine Vergleichbarkeit oder Klassifizier- und Abgrenzbarkeit verschiedener Techniken erweist sich oft als schwierig. Beispielsweise behindern verschiedene zu Grunde liegende Abstraktionsebenen einzelner Techniken den direkten Vergleich. Einige Techniken bilden Werte aus einem Wertebereich *W* in den gleichen oder einen anderen Wertebereich *W'* aus Zahlen und/oder Symbolen ab, Kodierungen aus *W* nach $C \subseteq \{0, 1\}^*$, wieder andere Techniken sind ausschließlich auf Bitebe-

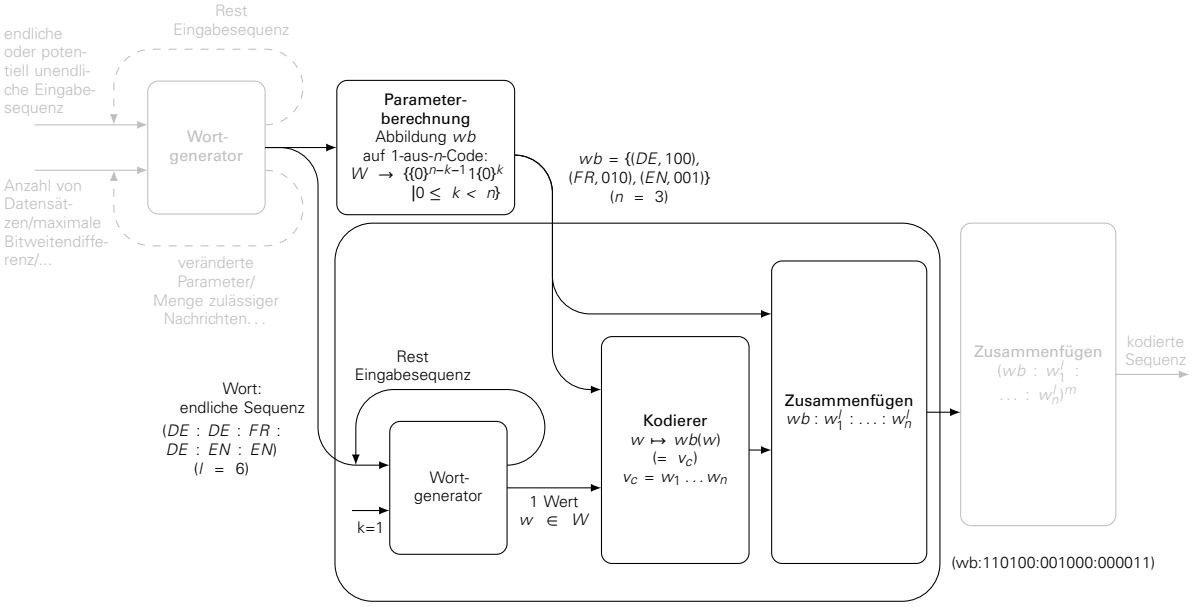


Abbildung 3.6: Bitvektorverfahren

ne angesiedelt. Es ist müßig, weiterhin zwischen einer rein mathematischen Ebene z.B. der natürlichen Zahlen bei FOR und einer Datentypebene zu unterscheiden, bei der alle Zahlen z.B. kleiner 2^{32} sind. Tabelle 3.1 zeigt einen Einordnungsversuch der sechs beschriebenen sehr allgemeinen Muster sowie weitere in [Reg81] vorgestellte Techniken, die etwas unmotiviert nebeneinander gestellt und beschrieben werden und sich schwer miteinander vergleichen lassen. Manche Ideen und Algorithmen lassen sich nicht eindeutig zuordnen. DICT als sehr allgemeines Schema kann zum Beispiel lange Strings durch kürzere ersetzen ($W \rightarrow W$) oder eine binäre Kodierung von Symbolen und beinhalten ($W \rightarrow C$). Möglich ist auch eine Auffassung explizit auf Bitebene ($C \rightarrow C'$), wobei kein Wissen über eine Semantik benötigt wird, wohingegen FOR und DELTA natürliche Zahlen auf andere natürliche Zahlen bzw. Integerwerte auf andere Integerwerte abbilden und diese hohe Abstraktionsebene ($W \rightarrow W'$) immer von Bedeutung ist und nicht abgespalten werden kann. Man kann versuchen innerhalb der drei Abstraktionsebenen Techniken als Verallgemeinerungen bzw. Spezialisierungen anderer aufzufassen, was durch Einrückungen zum Ausdruck gebracht ist.

Unter *Ausnutzung geordneter Daten* versteht man in [Reg81] Sequenzen von sortiert vorliegenden Zeichenketten. Dabei wird die Anzahl der führenden Symbole gespeichert, die sich nicht vom Vorgänger unterscheiden sowie alle danach folgenden. In der Sequenz (Datenkompression:Datenmenge) würde das erste Wort unkomprimiert gespeichert und nicht verändert werden, das Wort Datenmenge als (5,menge). *Mustersubstitution* ist ein Wörterbuchverfahren, welches Muster (endliche Sequenzen) in potentiell unendlichen Sequenzen aus Zeichen $m \in W^* \setminus \{m'\}$ durch kürzere endliche Sequenzen m' ersetzt. Bei der *kompakten Notation* ist aus dem Kontext

Tabelle 3.1: Abstraktionsebenen verschiedener Komprimierungstechniken

$W \rightarrow W'$	$W \rightarrow C$	$C \rightarrow C'$
DICT	DICT	DICT
RLE	BV	RLE
FOR	Variable-Längen-Kodierungen	Symbolunterdrückung
DELTA	Beschränkte Variable-Längen-Kodierungen	Nullenunterdrückung
Ausnutzung geordneter Daten	Dezimal-zu-Binär-Konversion	Binary Packing
Mustersubstitution		
Kompakte Notation		
Symbolunterdrückung		

bekannt, welche Werte eines Wertevorrates überhaupt vorkommen können und welche nicht. Damit kann der Wertebereich W abgebildet werden auf einen anderen Wertebereich W' mit $|W'| < |W|$. Bei der *Variable-Längen-Kodierung* werden häufiger vorkommende Werte aus einem Wertebereich W auf kürzere Codewörter aus C abgebildet als seltener vorkommende Werte. Bei der *beschränkten Variable-Längen-Kodierung* ist dies ebenso der Fall, nur sind für Codewörter aus $C \subset \{0, 1\}^*$ nur bestimmte Längen erlaubt.

Von Kompression kann man nur sprechen, wenn man ein Kompressionsverhältnis, d.h. das Verhältnis zwischen Datengröße ohne Kompression zu Datengröße mit Kompression angeben kann. Möglich ist das auf der Ebene $C \rightarrow C'$, also bei allen Verfahren auf Bitebene, oder wenn nach einem Standardverfahren kodierte Daten auf höherer Ebene vorverarbeitet und anders kodiert werden ($C \rightarrow W \rightarrow W' \rightarrow C'$). Bei vielen konkreten Algorithmen basiert die eigentliche Kompression vorverarbeiteter Daten auf Binary Packing.

Tabelle 3.2 zeigt die Module aller sechs betrachteten Muster im Vergleich. FOR und Bitvektoren sind die einzigen Muster, die im Normalfall semiadaptiv sind. FOR und DELTA gleichen sich im Kodierer, unterscheiden sich aber beim Zusammenfügen. Kodierer können bei metrischen Eingabewerten Ausgaben berechnen, bei nicht metrischen Werten ist ein definiertes Mapping zwingend. Die Art des Zusammenfügens ist der Übersichtlichkeit und Verständlichkeit halber weniger datailliert ausgeführt als in den einzelnen Kapiteln.

Tabelle 3.2: Vergleich der Module der verschiedenen Muster

		FOR	DELTA	DICT	Bitvektoren	Einfaches RLE	Symbolunterdrückung mit RLE
Wortgenerator		statisch/ datenabhängig			statisch/ datenabhängig		
Parameter-berechnung		nicht adaptiv			nicht adaptiv		
Kodierer/ Rekursion	Wortgenerator	statisch	statisch	statisch/ datenabhängig	statisch	datenabhängig	statisch/ datenabhängig
	Parameter-berechnung	-	adaptiv	-	-	-	-
	Kodierer	Berechnung		Berechnung/ Mapping	Mapping	Berechnung (+Mapping)	Berechnung (+Mapping)
	Zusammenfügen	$d : (v_c)^n$	$(v_c)^n$	$(v_c)^n$	$d : (v_c)^n$	$(d : v_c)^n$	$(d : v_c)^n$, $d^n : (v_c)^n$, $d : (v_c)^n$
Zusammenfügen		v_c^n			v_c^n		

3.8 ZUSAMMENFASSUNG

Die sechs vorgestellten Techniken haben inhärente Merkmale, die allerdings keine abgrenzenden Klassifikationsmerkmale darstellen. Auch Unter- und Überordnungen gibt es nur begrenzt. Mit der sehr allgemein gehaltenen Definition der Wörterbuchkompression fallen natürlich alle anderen Techniken, da sie einen Kodierer besitzen, unter Wörterbuchalgorithmen, da das einzige Merkmal des Wörterbuchmusters die Existenz eines Kodieres ist, der ein Element aus einem Wertevorrat als Eingabe nimmt und es auf ein Element des gleichen oder eines anderen Wertevorrates abbildet. Symbolunterdrückung und Lauflängenkodierung haben viele gemeinsame Merkmale (vgl. Abb. 3.4), wenn bei der Symbolunterdrückung Läufe des ausgezeichneten Symbols mit RLE komprimiert werden und dabei die Lauflänge als Deskriptor ausgegeben wird oder in die Berechnung des Deskiptors eingeht. Ebenso haben Frame of Reference und Differenzkodierung mit dem identischen Kodierer eine große Gemeinsamkeit. Sinnvoller als die Postulation von Kategorien ist die Betrachtungsweise als definierte Muster, sodass bei einer Modularisierung eines konkreten Algorithmus sichtbar ist, welche Muster an welcher Stelle auftreten. Die verschiedenen Muster definieren häufig nur bestimmte Teile der Modularisierungsschemas, sodass es viele Möglichkeiten für das Auftreten mehrerer Muster gibt, was sich auch im nächsten Kapitel bei der Betrachtung konkreter Algorithmen zeigt.

4 KONKRETE ALGORITHMEN

Die im letzten Kapitel vorgestellten Muster finden sich in verschiedenen Kompressionsalgorithmen, oft auch kombiniert oder miteinander verwoben, wieder. Abbildung 4.1 zeigt eine Übersicht über verschiedenste Verfahren und Verfahrensfamilien, die oft auf dem gleichen Grundgedanken aufbauen. *Binary Packing* (Abschnitt 4.1) liegt allen *FOR*- (Abschnitt 4.2) und *Simple*-Algorithmen zugrunde. Viele FOR-Algorithmen können in *Adaptive-FOR*- (Abschnitt 4.3) und *Patched FOR*-Methoden (Abschnitt 4.4) eingeordnet werden. Bei den Simple-Algorithmen (Abschnitt 4.5) wird eine variable Anzahl von Integerwerten mit 32 bzw. 64 Bits kodiert. Bei *byteorientierten Kodierungen* (Abschnitt 4.6) werden Integerwerte jeweils durch eine feste Anzahl von Bytes repräsentiert. Weiterhin werden in Abschnitt 4.7 verschiedene Wörterbuchalgorithmen vorgestellt.

4.1 BINARY PACKING

Binary Packing [LB12] (in der Literatur auch unter anderen Bezeichnungen wie z.B. PackedBinary [AM05] zu finden) ist eine Technik, bei der eine endliche Anzahl binär kodierter Integerwerte mit der gleichen, möglichst geringen Bitweite kodiert wird. Da dies durch die Entfernung führender Nullen gelingt, lassen sich in den Modularisierungen aller Algorithmen, die Binary Packing nutzen, das Symbolunterdrückungs- und das RLE-Muster finden. Im Kompressionsschema ist eine Rekursion vonnöten, da zunächst endliche Sequenzen vorliegen müssen, um eine gemeinsame Bitweite *bw* für alle Werte festzulegen, sofern nicht der Kontext Informationen über einen nach oben beschränkten Wertebereich liefert. Während der Rekursion werden alle Werte binär mit Bitweite *bw* kodiert. An dieser Stelle soll der in weiteren Definitionen verwendete Begriff der *signifikanten Bits einer binären Repräsentation* eines Wertes eingeführt werden.

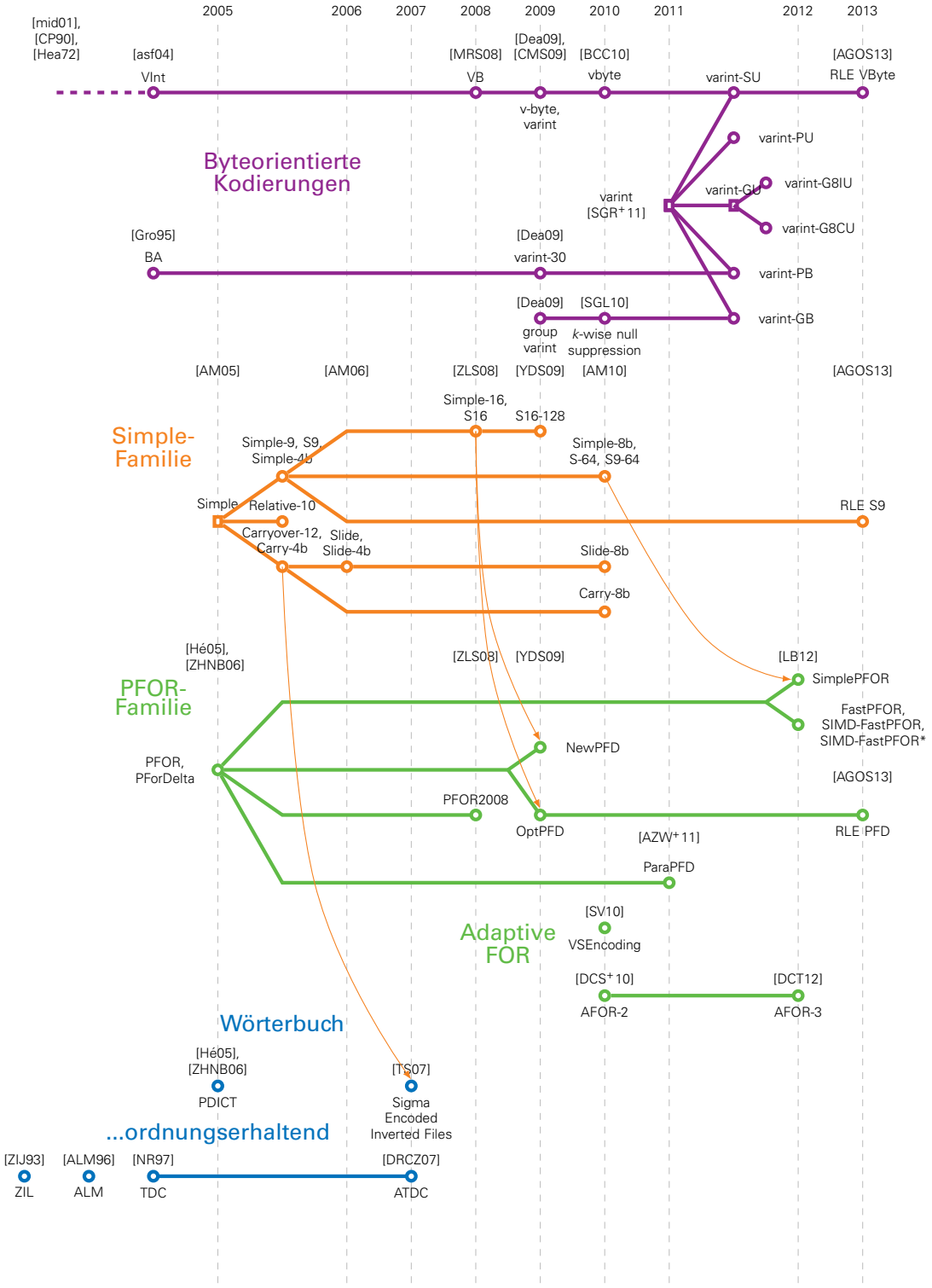


Abbildung 4.1: Übersicht über einige Kompressionsalgorithmen

Definition 4.1.1 (*Signifikante Bits einer binären Repräsentation*)

Eine binäre Repräsentation w eines Wertes ist ein Element $w \in \{0, 1\}^{32}$. *Die binäre Repräsentationen* 0^{32} *hat keine signifikanten Bits. Alle anderen binären Repräsentationen sind Elemente der Menge* $0^n1\{0, 1\}^{31-n}$ *und zeichnen sich durch n führende Nullen aus. Alle Bits außer den führenden Nullen sind die signifikanten Bits einer binären Repräsentation eines Wertes.*

Abbildung 4.2 zeigt das Kompressionsschema für Binary Packing. Der erste Wortgenerator erzeugt endliche Sequenzen ($m_1 : \ldots : m_k$) aus mit 32 Bits kodierten Integerwerten. Wie genau diese Sequenzen für das Binary Packing festgelegt werden und wie genau der Wortgenerator definiert ist, ist für das Binary-Packing-Muster nicht wichtig. Da seine Eingabe ein Datenstrom ist, muss er wie alle Wortgeneratoren zu Beginn des Kompressionsschemas rekursiv sein. Der Wert k kann von Sequenz zu Sequenz verschieden sein. Im Modul der Parameterberechnung wird eine gemeinsame Bitweite für die Werte $m_1, \ldots, m_k$ festgelegt, diese ist mindestens so groß wie die Bitweite, die für den größten der Werte nötig ist. Der Wert l_i bezeichnet im Schema die Anzahl der führenden Nullen. Feste Parameter als Eingabe können dazu dienen, weitere Bedingungen festzulegen, beispielsweise eine Auswahl an möglichen Bitweiten vorzugeben. In die Rekursion gehen die berechnete Bitweite *bw* sowie die endliche Sequenz ($m_1 : \ldots : m_k$) ein. Der Wortgenerator innerhalb der Rekursion erzeugt einzelne Integerwerte zu 32 Bits. Der Kodierer berechnet den komprimierten Wert, indem er abhängig von der festgelegten Bitweite eine bestimmte Anzahl führender Nullen entfernt. Im Modul des Zusammenfügens innerhalb der Rekursion wird die Bitweite als gemeinsamer Deskriptor mit allen k Werten konkateniert.

4.2 FOR MIT BINARY PACKING

Zumeist werden für die Definition des FOR weitere Einschränkungen als ganz selbstverständlich gesehen. Möglicherweise soll die Eingabe endlich sein, damit der Referenzwert als kleinster Wert bestimmt werden kann. Im Normalfall wird man die einzelnen Werte dann mit Binary Packing binär kodieren, was wiederum im Schnittbereich zwischen Nullenunterdrückung und Lauflängenkodierung liegt. Diese einschränkendere Definition des FOR zeigt Abb. 4.3. Zunächst wird für jede endliche Sequenz ($m_1 : \ldots : m_k$) ein Referenzwert sowie eine mögliche Bitweite als Parameter berechnet. Will man keine negativen Werte kodieren, darf der Referenzwert m_{ref} maximal genauso groß sein wie der kleinste Wert $\inf(\{m_i | 0 < i \leq k\})$ innerhalb der endlichen Sequenz. Die Bitweite muss so groß sein, dass der die Differenz aus dem größten Wert der endlichen Sequenz und dem Referenzwert m_{ref} damit kodierbar ist. Innerhalb der Rekursion wird dann die Differenz aus Wert und Referenzwert mit der Bitweite *bw* kodiert. Bitweite und Referenzwert müssen natürlich mit gespeichert werden. Die Operationen des FOR und des Binary Packing liegen auf zwei verschie-

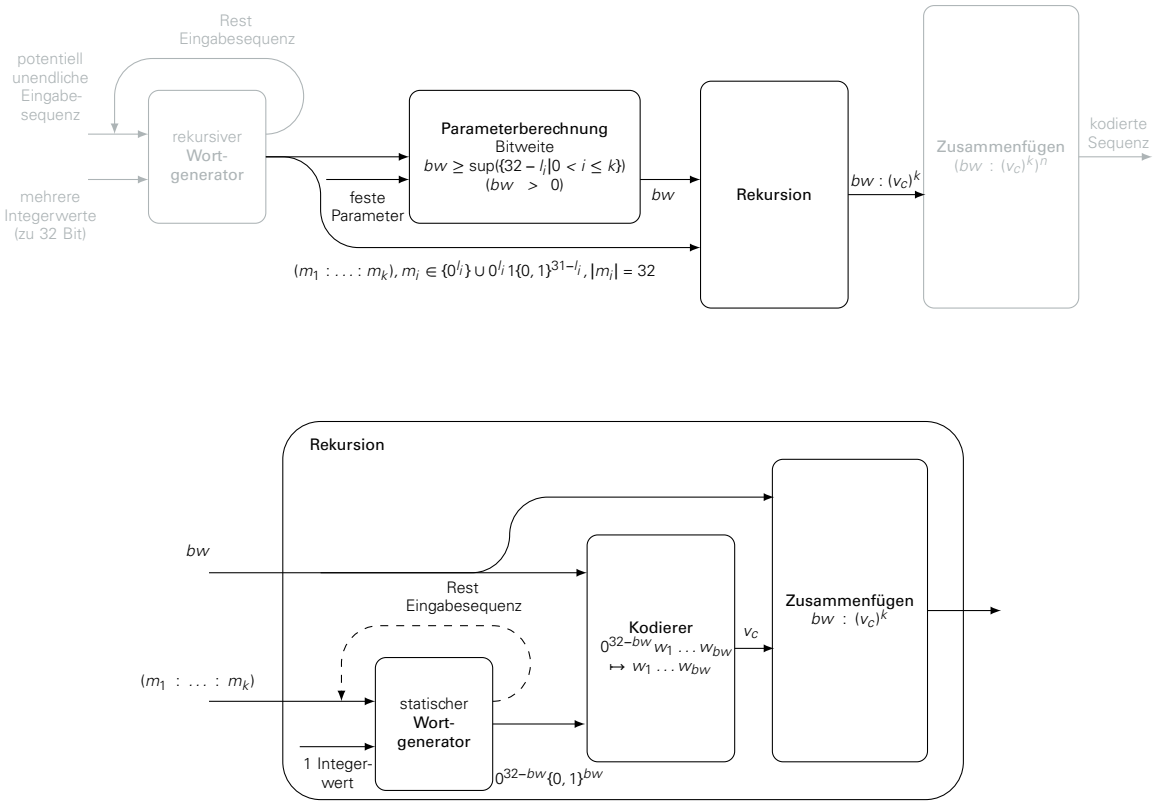


Abbildung 4.2: Kompressionsschema Binary Packing

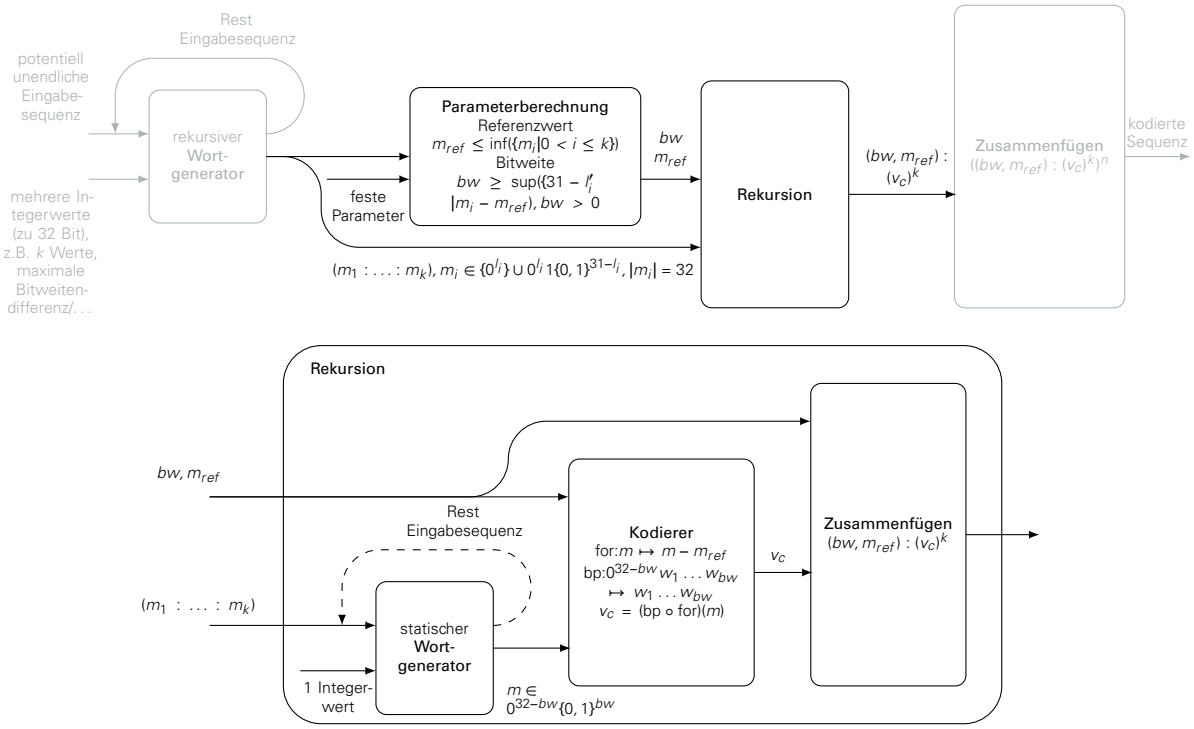
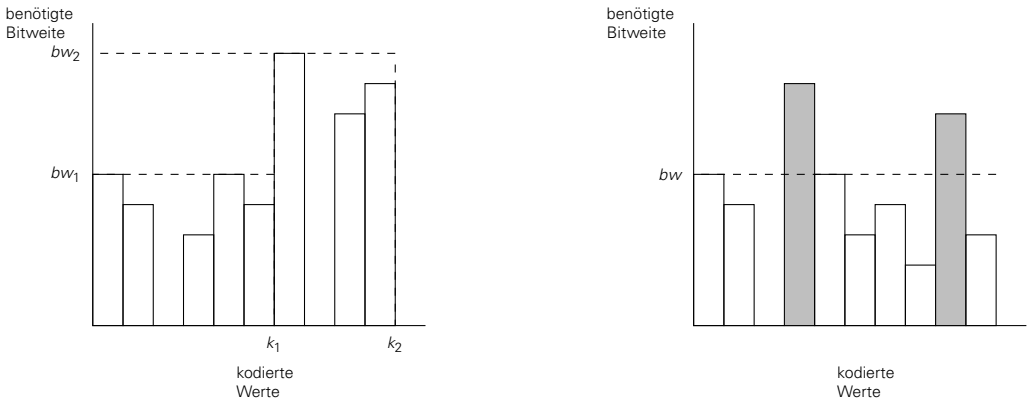


Abbildung 4.3: Einfaches semiadaptives FOR-Kompressionsschema mit Binary Packing

denen Abstraktionsebenen. FOR kann auf Ebene der natürlichen Zahlen betrachtet werden. Binary Packing nicht auf höherer als auf Bitebene. Die eigentliche und messbare Kompression beinhaltet das Binary Packing. FOR mit Binary Packing beinhaltet das Muster DICT wegen des Vorhandenseins eines Kodieres, FOR sowie RLE und Symbolunterdrückung wegen der unterdrückten Läufe aus Nullen, deren Läuflängeninformation aus der Bitweite *bw* und dem Kontextwissen, dass Integerwerte unkomprimiert mit 32 Bits gespeichert werden, besteht.

4.3 ADAPTIVE FOR UND VSENCODING

Die Kompressionsrate (als Verhältnis von Datengröße unkomprimierter Werte zu Datengröße komprimierter Werte) für eine endliche Sequenz wird bei FOR mit Binary Packing umso größer, je geringer die Differenzen der Bitweiten zwischen den einzelnen Werten sind. Statt einer festen Anzahl *k* von Werten mit gemeinsamer Bitweite kann die Kompressionrate oft trotz zusätzlichen Overheads deutlich verbessert werden, wenn *k* datenabhängig bestimmt wird (siehe Abb. 4.4a). Zwei Komressionsalgorithmen werden hierzu kurz vorgestellt, die im Grunde genommen nicht in die Kategorie FOR wie hier definiert zählen, sondern nur Binary Packing realisieren. AFOR-2 [DCS+10], modularisiert dargestellt in Abbildung 4.5, betrachtet zunächst mit einer gegebenen Fenstergröße eine feste Anzahl von Werten (z.B. 32 Werte). Der Wort-



(a) Adaptive FOR (b) PFOR

Abbildung 4.4: Datenunterteilung und Bitweiten bei Adaptive FOR und PFOR

generator innerhalb der ersten Rekursion erhält als Parameter mögliche Framelängen, Teiler der Fenstergröße (z.B. 32, 16 und 8 Werte). Es ist ein kombinatorisches Optimierungsproblem, nun die beste Zerlegung der Werte zu finden. Der zweite Wortgenerator in der ersten Rekursion findet aus den 6 Möglichkeiten m^{32}, (m^{16}:m^{16}), (m^{16}:m^8:m^8), (m^8:m^{16}:m^8), (m^8:m^8:m^{16}), (m^8:m^8:m^8:m^8) gemäß eines in [DCS+10] beschriebenen Algorithmus, der eine sinnvolle Berechnungsreihenfolge vorgibt, die optimale Wahl der Unterteilung. Auch hier gibt es einen starken Zusammenhang zwischen der optimalen Unterteilung, die im Wortgenerator berechnet wird, und den passenden Bitweiten, die von der Parameterberechnung ausgegeben werden. Jeder Frame wird innerhalb der 2. Rekursion mit Binary Packing komprimiert. Der in der ersten Rekursion in der Parameterberechnung festgelegte Bit Frame Selector ist ein im Paper nicht näher erläuterter Deskriptor, der zumindest Bitweite *bw* und Anzahl *k* der Werte kodiert.

Die Modularisierung von VSEncoding [SV10] folgt dem gleichen 3-stufigen Schema wie AFOR-2 mit mehreren inhaltlichen Unterschieden. VSEncoding hat keine Vorgaben für den Wortgenerator auf höchster Ebene und bietet für den 2. Wortgenerator einen Optimierungsalgorithmus für alle denkbaren Unterteilungen einer endlichen Sequenz. Wie auch bei AFOR-2 existiert in Rekursion 1 eine hohe Modulkopplung zwischen Wortgenerator und Parameterberechnung. Eine endliche Sequenz wird in Rekursion 1 in m Teilsequenzen untergliedert. Eine Teilsequenz $m_1 \ldots m_{k_i}$ zeichnet sich durch eine Bitweite bw_i und die Anzahl ihrer Werte k_i aus. Bitweiten werden z.B. mit Elias-Gamma kodiert, Anzahlen z.B. unär, Werte binär gepackt mit Bitweite bw_i. Somit ergeben sich für jede Teilsequenz berechenbare Kosten, die für die Gesamtsequenz in Abhängigkeit von der Unterteilung einfach aufsummiert werden können. VSEncoding basiert auf einem Algorithmus, der die hinsichtlich der Kompressionsrate optimale Unterteilung für endliche Sequenzen von Integerwerten bestimmt und nicht wie AFOR-2 aus einer Vorauswahl möglicher Unterteilungen ein Optimum findet.

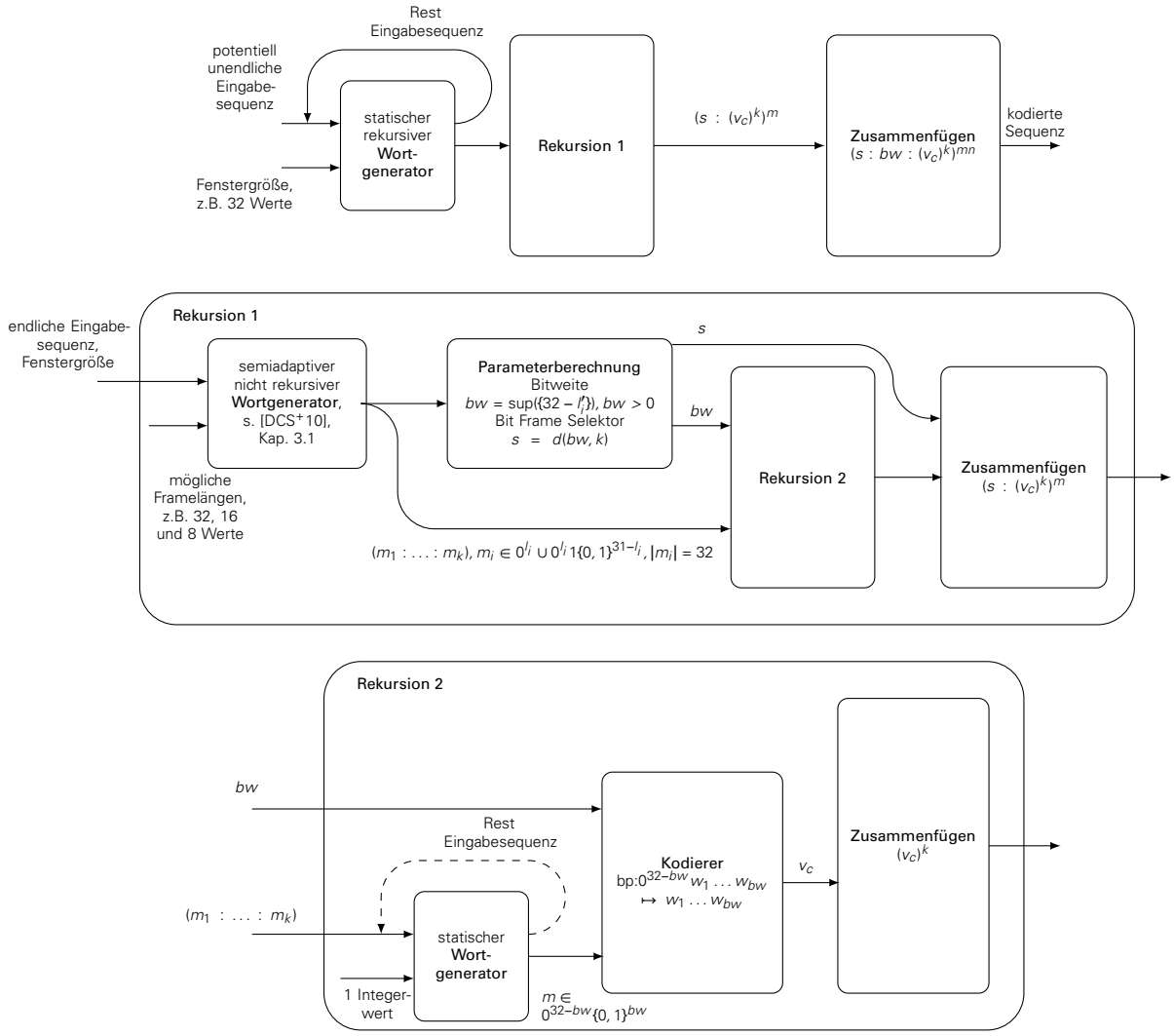


Abbildung 4.5: Kompressionsschema AFOR-2

4.4 PFOR-ALGORITHMEN

PFOR-Algorithmen widmen sich ebenso wie AFOR der Frage, welche Werte mit einer gemeinsamen Bitweite binär gepackt werden können, um eine möglichst gute Kompressionrate zu erhalten. Im Gegensatz zu AFOR werden bei PFOR-Algorithmen einzelne Werte als Ausnahmen deklariert, so dass ein Großteil der Werte einfach binär gepackt wird (siehe Abb. 4.4b), die Ausnahmen jedoch anders behandelt und physisch an anderer Stelle gespeichert werden. Gerade diese Algorithmenfamilie, in der die Werte umorganisiert werden, benötigt bei ihrer Modularisierung das Splitmodul. Nicht alle PFOR-Algorithmen beschäftigen sich wirklich mit einem von 0 verschiedenen Referenzwert. Deswegen taucht das FOR-Muster in den Modularisierungsschemata in diesem Kapitel häufig nicht auf. Allen gemein ist das Binary Packing und damit das Muster der Symbolunterdrückung, welches sich in vielen Kodierern findet, und das Muster der Lauflängenkodierung.

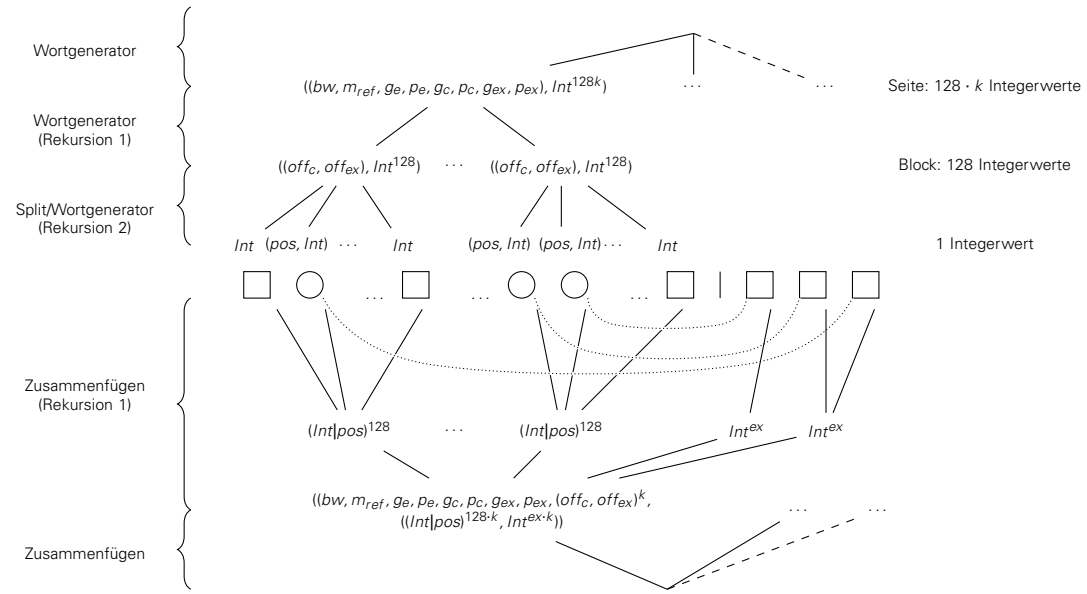
4.4.1 PFOR UND PFOR2008

In [Hé05] und [ZHNB06] wird PFOR, der erste Algorithmus mit dieser komplexen Idee, erläutert. Der Verständlichkeit des Modularisierungsgedankens wegen soll zunächst die Struktur der komprimierten Daten anhand von Abbildung 4.6 erläutert werden. Pro Seite ($128 \cdot k$ Integerwerte) wird anhand einer Stichprobe aus N Werten eine gemeinsame Bitweite bw und und ein gemeinsamer Referenzwert m_{ref} festgelegt, so dass ein Teil der Werte mit den größten Abweichungen nicht mit der gegebenen Bitweite kodiert werden kann und als Ausnahme deklariert werden muss. Für die weitere Verarbeitung werden die Werte in k Blöcke zu 128 Integerwerten unterteilt. Auf Implementierungsebene wird, um teure Verzweigungen zu vermeiden, jeder Block wie folgt mit zwei Schleifen kodiert. Mit einer Iteration über alle 128 Werte wird eine Liste der Länge 128 mit kodierten Werten und eine Liste mit Ausnahmepositionen gefüllt. Die zweite Schleife iteriert über die Ausnahmepositionen und füllt dabei zum einen eine Liste für Ausnahmen, welche mit 32 Bits kodiert, also nicht komprimiert werden. Zum anderen wird die Liste mit kodierten Werten an den Ausnahmepositionen mit dem Abstand zur nächsten Ausnahmeposition überschrieben. Da die in der ersten Schleife kodierten Werte mit Bitweite bw gepackt werden, stehen auch für die Kodierungen der Abstände zur nächsten Ausnahmeposition nur bw Bits zur Verfügung. Ist der Abstand größer als 2^{bw} Werte, muss eine Ausnahme erzwungen werden. Für jeden Block gibt es also zwei Arten von kodierten Werten, eine Liste L_1 mit binär gepackten Werten (in Abbildung 4.6 als Rechtecke dargestellt) und eingebetteten Positionsabgaben (in Abbildung 4.6 als Kreise dargestellt) sowie eine Liste L_2 mit unkomprimierten Ausnahmen. Für jeden Block ist ein sogenannter entry point als Deskriptor gegeben, ein Tupel (off_c, off_{ex}) aus einem Zeiger zur ersten Ausnahmeposition in L_1 und einer zur ersten Ausnahme in L_2. Jede Positionsangabe pos (bzw.

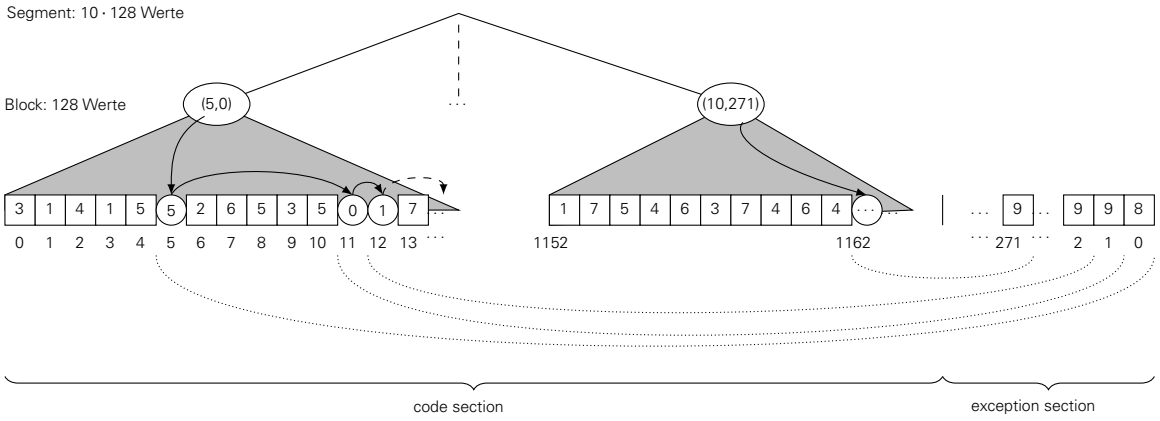
jeder Abstand) für Ausnahmen in L_1 kann als eine Art Deskriptor für einen Ausnahmewert betrachtet werden. Auf Segmentebene werden sowohl alle k Listen vom Typ L_1 konkateniert und alle k Listen vom Typ L_2 zusammengefügt.

Abbildung 4.6b zeigt ein Beispiel für den Aufbau eines Segments. Ein Segment besteht aus $10 \cdot 128$ Werten. Im Beispiel sind das die ersten Ziffern von π. Die Bitweite für das gesamte Segment ist mit $bw = 4$ und der Referenzwert mit $m_{ref} = 0$ festgelegt, so dass die Werte von 0 bis 7 normal mit 4 Bits kodiert werden und die Werte 8 und 9 Ausnahmen sind. Die code section enthält alle normal kodierten Werte und Positionsinformationen für die Ausnahmen. Alle Ausnahmen für das gesamte Segment stehen danach in der exception section. Ein Segment wird im Beispiel in 10 Blöcke unterteilt. Jedem Segment ist ein Tupel (off_c, off_{ex}) als Deskriptor zugeordnet. Das erste Element des Tupels ist die Position des ersten Ausnahmewertes im Block. Im ersten Block befindet sich die erste Ausnahme an Position 5, somit ist $off_c = 5$. Das zweite Element des Tupels ist der korrespondierende Offset in der exception section. Für den ersten Block ist das der Wert $off_{ex} = 0$, für den 10. Block der Wert $off_{ex} = 271$, weil in den neun Blöcken davor bereits 271 Mal entweder die Werte 8 und 9 als Ausnahmen vorkamen oder Ausnahmen aufgrund zu hoher Abstände Ausnahmen erzwungen werden mussten.

Die Modularisierung ist in Abb. 4.7 dargelegt. Auf oberster Ebene wird der Datenstrom in Segmente zu $k \cdot 128$ Werten unterteilt, ein Referenzwert m_{ref} sowie die Bitweite bw berechnet. Rekursion 1 befasst sich mit genau einem Segment. Dieses wird mit dem Wortgenerator in k Blöcke unterteilt. Rekursion 2 entspricht der Blockebene. Das Splitmodul teilt anhand von Bitweite bw, Referenzwert m_{ref} und Abstand zur letzten Ausnahmeposition Δ einzelne Integerwerte in normale Daten und Ausnahmen auf. Ein Wert wird im Splitmodul der Gruppe der Ausnahmen zugeordnet, wenn er kleiner als der Referenzwert ist, die Differenz zum Referenzwert nicht nur mit der Bitweite bw kodiert werden kann oder der Abstand Δ zur letzten Ausnahme zu groß ist, um ihn mit bw Bits zu kodieren. Da normale Daten und Abstände zwischen den Ausnahmepositionen mit 3 Bits kodiert werden sollen, müssen bei größeren Abständen als acht Positionen Ausnahmen erzwungen werden. Der Kodierer für normale Daten gleicht dem des FOR mit Binary Packing (siehe Ab. 4.3). Die *ex* Ausnahmewerte werden nicht komprimiert, sondern mit 32 Bits kodiert. Positionsangaben werden so verarbeitet, das die Information über den ersten Ausnahmewert als entry point (off_c, off_{ex}) und sonst die Differenzen Δ zwischen den Ausnahmepositionen ausgegeben werden. Auf Segmentebene werden im Modul des Zusammenfügens in der entry point section alle k entry points konkateniert, in der code section alle k Blöcke mit normalen Daten und Ausnahmepositionsdifferenzen sowie in der exception section alle Ausnahmen. Im Modul des Zusammenfügens kommen im Header noch die Größen (*g*)- und Positionsinformationen (*p*) für die entry point section (*ep*), code section (*c*) und exception section (*ex*) hinzu. Das Kompressionsschema unterstützt leider nicht die Berechnung von Deskriptoren für mehrere Werte, welche praktisch erst nach der Kodierung berechnet werden können, wie z.B. Positions- und Größen-



(a) Datenorganisation



(b) Beispiel für die Datenorganisation innerhalb eines Segments

Abbildung 4.6: Hierarchische Datenorganisation bei PFOR und PFOR2008

angaben.

PFOR2008 [ZLS08] folgt dem gleichen Modularisierungsschema. Während in [Hé05] und [ZHNB06] für PFOR vorgeschlagen wird, Bitweiten von mindestens 8 zu nutzen, sind für PFOR2008 explizit alle Bitweiten ≤ 32 erlaubt, was in der Modularisierung in der Parameterberechnung auf oberster Ebene ausgedrückt wird. Außerdem werden Ausnahmen pro Seite nicht unbedingt mit einer Bitweite $bw_{max} = 32$ gespeichert, sondern, wenn möglich, mit 8 oder 16 Bits. Praktisch würde diese Bitweite bestimmt werden, wenn alle k Ausnahmelisten konkateniert sind. Im Modularisierungsschema findet sich diese Information genauso in der Parameterberechnung auf oberster Ebene. Die sich von PFOR unterscheidenden Module sind in Abb. 4.8 grau hervorgehoben. Der ursprüngliche PFOR-Algorithmus ist der einzige, in dessen Literaturangaben der Gedanke des FOR und damit eines Referenzwertes eine Rolle

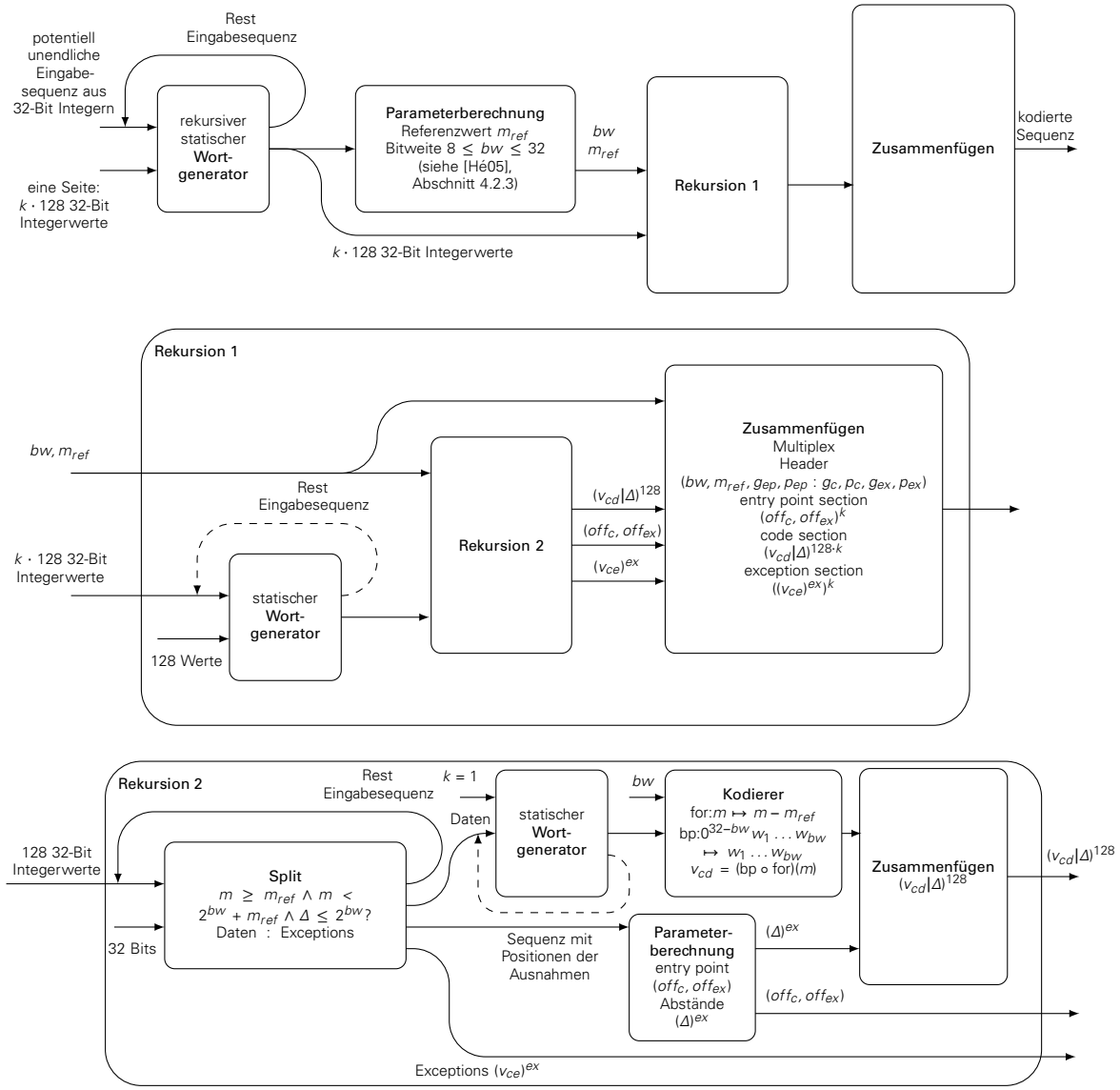


Abbildung 4.7: Kompressionsschema PFOR

spielt, vermutlich, weil die meisten Autoren davon ausgehen, dass die Eingabewerte bereits differenzkodiert und damit ausreichend klein sind um grundsätzlich 0 als Referenzwert zu wählen. Die folgenden Algorithmen sind meist so implementiert, dass sie Binary Packing, jedoch nicht zwangsläufig FOR beinhalten. Die Modularisierung würde sich nicht ändern, würde man sie zu echten FOR-Algorithmen variieren.

4.4.2 NEWPFD UND OPTPFD

NewPFD und OptPFD [YDS09] haben das Ziel, kleinere Werte mit entsprechend kleineren Bitweiten zu kodieren und wollen dabei erzwungene Ausnahmen vermeiden. Wird etwa beim normalen PFOR-Algorithmus die Bitweite $bw = 3$ festgelegt, so darf der Abstand zwischen den Ausnahmen maximal 8 Positionen betragen. Somit werden mehr als 12% der Werte als Ausnahmen deklariert, weil mindestens jeder 8. Wert eine Ausnahme sein muss. Außerdem sollen auch die Ausnahmen besser komprimiert werden. Abbildung 4.9 zeigt die Unterschiede in der Datenorganisation beider Algorithmen im Vergleich zu PFOR und PFOR2008 auf. Beide Algorithmen definieren Bitweiten pro Block statt pro Segment. Auch Ausnahmen werden blockweise separiert. Ausnahmepositionen werden in einer separate Liste abgelegt und komprimiert. Da normale Integerwerte, die keine Ausnahmen sind, mit einer bestimmten Bitweite *bw* kodiert werden, können die niedrigeren Bits von Ausnahmen zwischen den normal kodierten Werten gespeichert werden, höhere Bits werden separat komprimiert (jeweils dargestellt durch halbe Quadrate). Da die Segmentebene vollständig entfällt, kann zwei- statt dreistufig modularisiert werden (siehe Abb. 4.10). Auf oberster Ebene wird zumindest die Bitweite *bw*, aber auch Parameter wie die Anzahl der Ausnahmen berechnet. NewPFD wählt die Bitweite so, dass maximal 10% der Werte Ausnahmen sind, bei OptPFD hingegen ist nicht die Exception-, sondern die Kompressionsrate Optimierungskriterium. Somit unterscheiden sich beide Algorithmen im in Abbildung 4.10 grau markierten Modul der Parameterberechnung. Mögliche Eingrenzungen der Auswahl an Bitweiten gehören zu implementierungsspezifischen Details, die das Modularisierungsschema nicht tangieren. In [LB12] beispielsweise wurden die möglichen Bitweiten auf 1-15, 20 und 32 beschränkt, um die Komprimierungszeit zu verringern. In der Rekursion unterscheiden sich NewPFD und OptPFD von PFOR und PFOR2008, was sich anhand eines Beispiels (siehe Tab. 4.1), entnommen aus [LB12], besser erläutern lässt. Integerwerte, bestehend aus durch die gegebene Bitweite $bw = 3$ definierte Anzahl niedrigerer ($b_\downarrow$) und höherer Bits ($b_\uparrow$) werden genauso mit einem Splitmodul in normale Daten und Ausnahmen aufgeteilt und getrennt betrachtet. Normale Daten werden mit 3 Bits kodiert. Der Wortgenerator für Ausnahmen gibt zum einen die niedrigeren 3 Bits und zum anderen die verbleibenden höheren Bits aus. Die normalen kodierten Daten und die niedrigeren Bits der Ausnahmen werden unter Erhaltung der Reihenfolge in der code section zusammengefügt (Notation: $(v_c|b_\downarrow)^{128}$). Die höheren Bits $b_\uparrow$ der Ausnahmen (in Tab. 4.1 markiert) werden genauso wie die Abstände der Ausnahmepositionen (Δ) mit dem Algorithmus

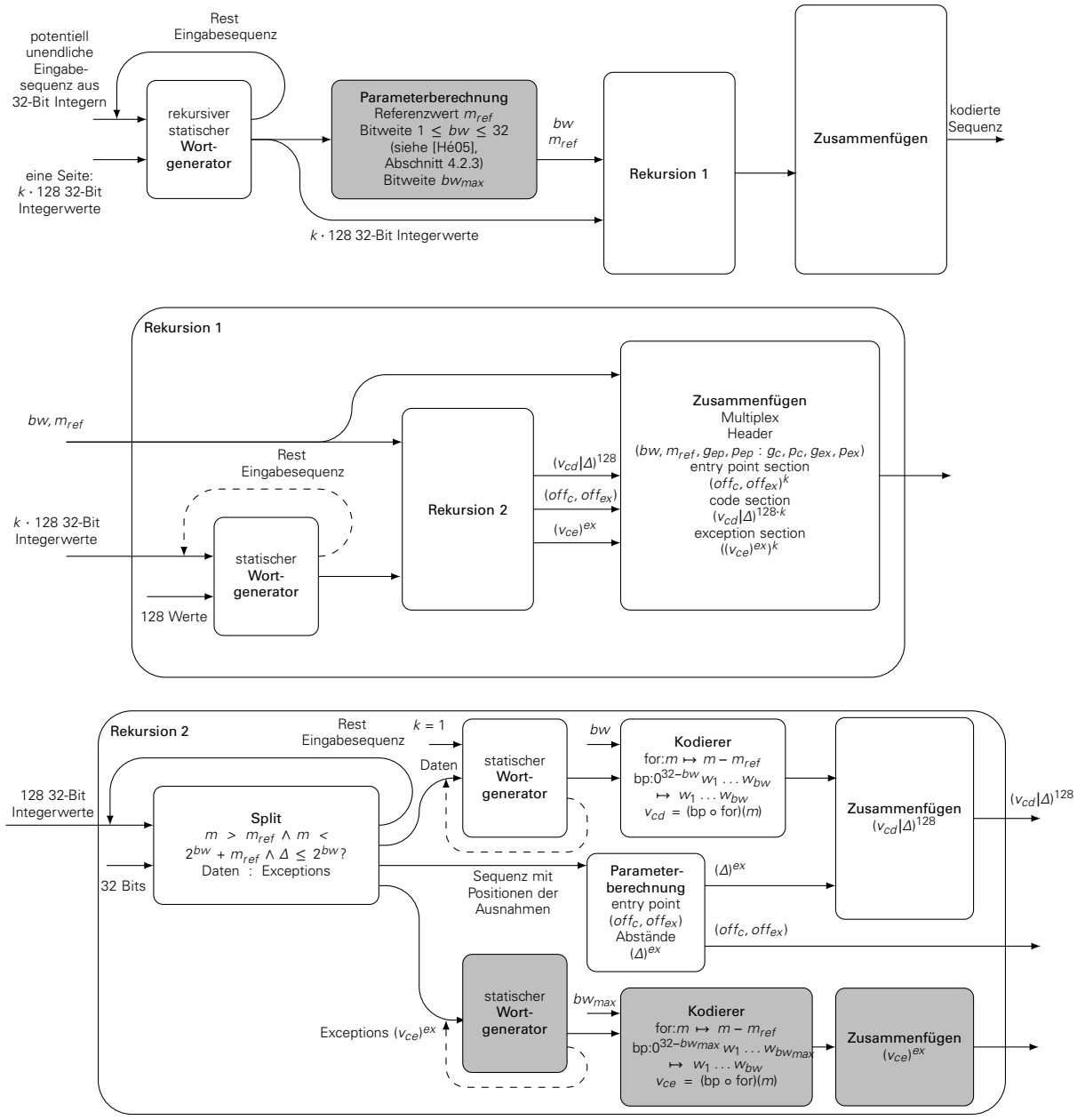


Abbildung 4.8: Kompressionsschema PFOR2008

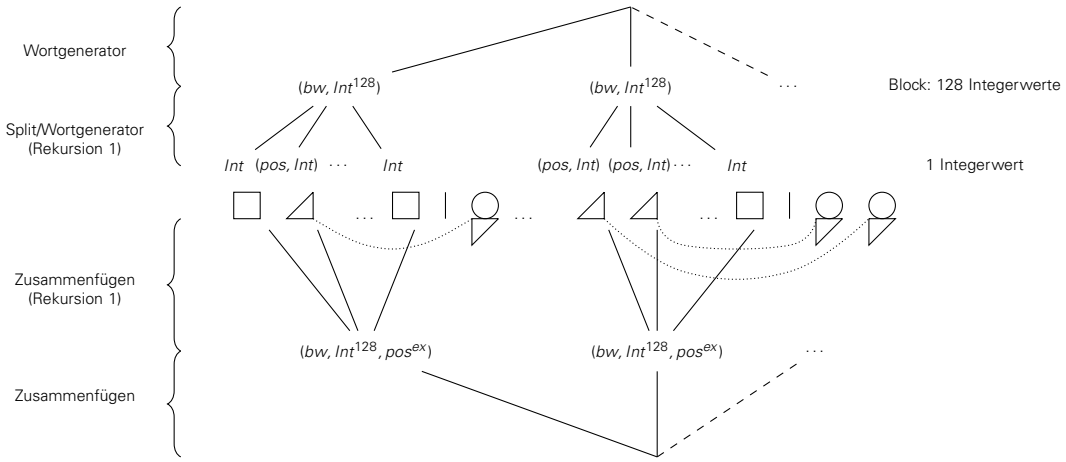


Abbildung 4.9: Hierarchische Datenorganisation bei NewPFD und OptPFD

Tabelle 4.1: Beispiel für die Anwendung der Algorithmen NewPFD und OptPFD mit Bitweite $bw = 3$

Position	Wert	Ausnahme	Δ	$b_\uparrow$	komprimierte Daten v_c bzw. $b_\downarrow$
0	2			$0^{26}000$	**010**
1	2			$0^{26}000$	**010**
2	1			$0^{26}000$	**001**
3	2			$0^{26}000$	**010**
4	38	$\star$	**4**	$\mathbf{0^{26}100}$	**110**
5	2			$0^{26}000$	**010**
6	1			$0^{26}000$	**001**
7	3			$0^{26}000$	**011**
8	2			$0^{26}000$	**010**
9	34	$\star$	**4**	$\mathbf{0^{26}100}$	**010**
10	2			$0^{26}000$	**010**
11	50	$\star$	**1**	$\mathbf{0^{26}110}$	**010**

Simple-16 (siehe Kapitel 4.5.2) komprimiert. Damit können die Abstände zwischen den Positionen mit bis zu 28 Bits kodiert werden. Ausnahmen müssen nicht erzwungen werden.

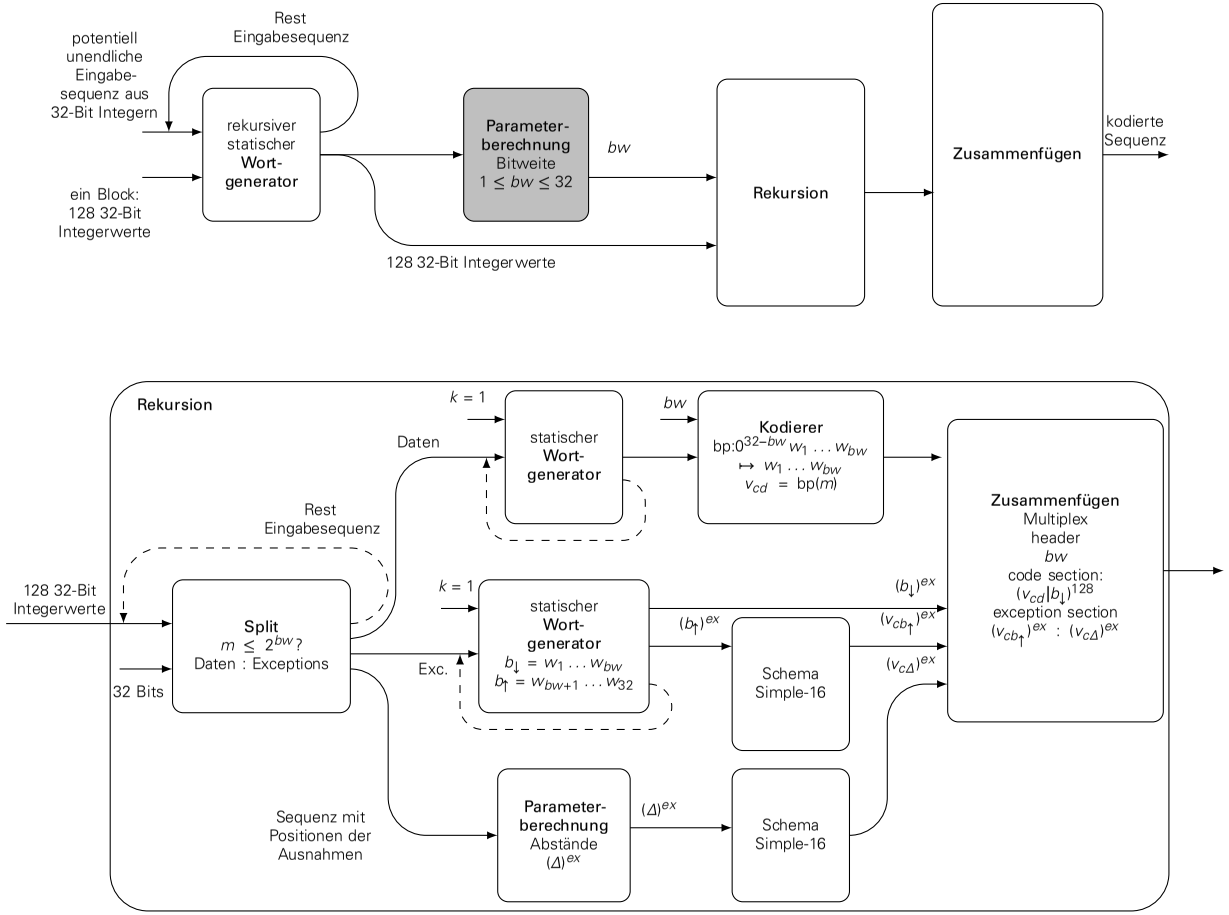


Abbildung 4.10: Kompressionsschema NewPFD und OptPFD

4.4.3 SIMPLEPFOR UND FASTPFOR

NewPFD komprimiert besser als PFOR, ist aber wesentlich langsamer. Die in [LB12] beschriebenen Algorithmen SimplePFOR und FastPFOR wurden mit dem Ziel entwickelt, ebenso gut die NewPFD zu komprimieren, jedoch mit der Geschwindigkeit von PFOR. In Abbildung 4.11 ist die Datenorganisation beider Algorithmen dargestellt. SimplePFOR und FastPFOR bestimmen wie auch NewPFD eine Bitweite pro Block, wodurch eine höhere Kompressionsrate erreicht wird als mit einer Bitweite pro Seite wie bei PFOR. Die Ausnahmen werden jedoch wie bei PFOR pro Seite separiert, um eine ähnlich hohe Kompressionsgeschwindigkeit wie bei PFOR zu erreichen. Das Modularisierungsschema muss demzufolge mindestens dreistufig sein, da die Segmentebene beim Zusammenfügen von Bedeutung ist. Positionsinformationen, höhere und niedrigere Bits werden analog zu NewPFD und OptPFD angeordnet. Die Positionen werden nicht als Differenzen, sondern mit 8 Bits kodiert, da bei 128 Werten nur die Positionen 0 bis 127 infrage kommen.

Abbbildung 4.12 zeigt das Kompressionsschema für beide Algorithmen. Auf Segmentebene werden keine Parameter berechnet. Die Beispieldaten zeigen einen Ausschnitt aus den Daten eines Segments. Auf Blockebene (Rekursion 1) wird als Parameter zunächst die maximale Bitweite bw_{max} für den größten der 128 Integerwerte bestimmt. Im Beispiel wird $bw_{max} = 6$ gesetzt. Die Berechnung der Bitweite $bw = 2$ pro Block, mit welcher normale Daten kodiert und wie bei NewPFD die Anzahl der niedrigeren Bits der Ausnahmen bestimmt werden, ist ein Optimierungsproblem nach den Speicherkosten. Diese setzen sich aus dem Platzbedarf für normale Daten ($128 \cdot bw$ Bit), für die Speicherung der ex Ausnahmepositionen mit jeweils 8 Bits ($8 \cdot ex$ Bits) und für die Speicherung der höheren Bits der Ausnahmen zusammen. Da der größte Wert mit einer Bitweite bw_{max} kodiert werden kann, belaufen sich die Kosten für Speicherung der höheren Bits der Ausnahmen auf $(bw_{max} - bw) \cdot ex$ Bits. Diese Rechnung dient im Grunde genommen nur als ausreichend genauer Überschlag, da die Ausnahmen nicht umkomprimiert gespeichert werden und die berechneten Werte nicht den tatsächlichen Speicherkosten entsprechen. In Rekursion 2, in der ein Block aus 128 Werten in einzelne Integerwerte unterteilt werden soll, dient wie gehabt ein Splitmodul zur Separation von normalen Daten und Ausnahmen. Niedrigere Bits der Ausnahmen und normale Daten werden genauso wie bei NewPFD reihenfolgeerhaltend zusammengefügt. Die Positionsangaben für Ausnahmen werden nicht differenzkodiert, sondern mit Bitweite 8 binär kodiert. Die höheren Ausnahmebits werden bei SimplePFOR und FastPFOR im in Abb. 4.12 grau markierten Teil auf verschiedene Weise kodiert. Während SimplePFOR mit Simple-8b kodiert, nutzt FastPFOR Arrays, die jeweils mit Ausnahmewerten $b_{\uparrow}$ der gleichen Bitweite $bw_{max} - bw$ gefüllt werden. Auf Blockebene können bereits sowohl die normal kodierten Daten und die niedrigeren Bits der Ausnahmen als auch die Ausnahmepositionsangaben zusammengefügt werden. Auf Segmentebene werden alle kodierten Daten und die niedrigeren Bits der Ausnahmen pro Segment konkateniert mit k Blockheadern, be-

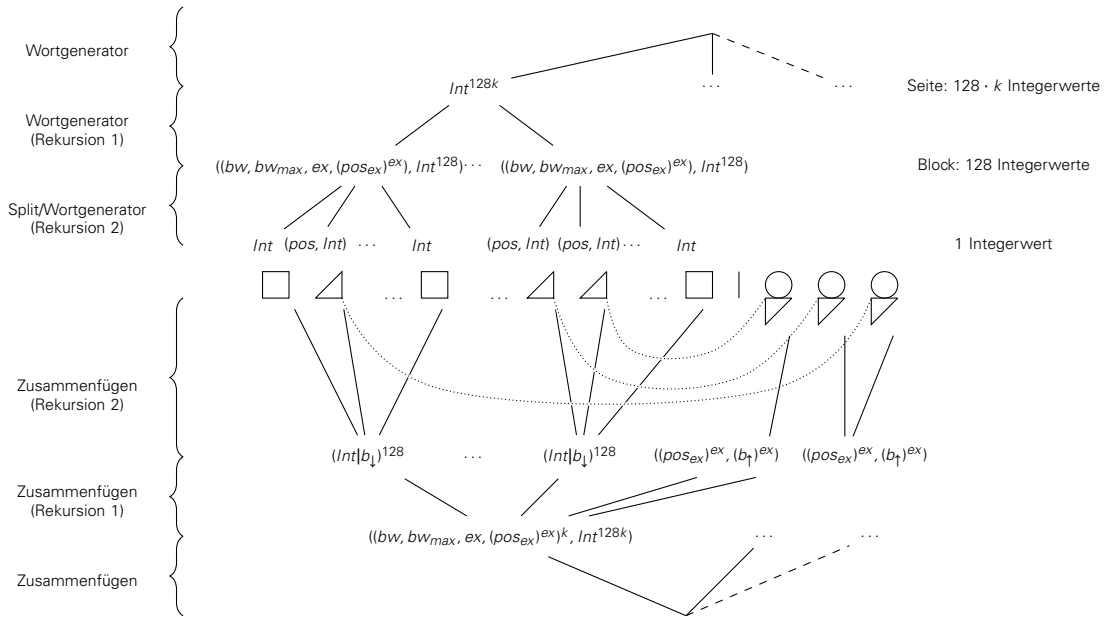


Abbildung 4.11: Hierarchische Datenorganisation bei SimplePFOR und FastPFOR

stehend aus Bitweite *bw*, maximaler Bitweite bw_{max}, Anzahl der Ausnahmen für den jeweiligen Block *ex* und den zusammengefügten Ausnahmepositionen. Es folgen die komprimierten Ausnahmen.

Möglich ist auch eine Modularisierung beider Algorithmen auf nur zwei Ebenen. Die Segmentebene kann prinzipiell entfallen, da keine Parameter für das gesamte Segment berechnet werden. Dann ist es Aufgabe das Moduls des Zusammenfügens, die Daten aus jeweils *k* Blöcken zu konkatenieren. An dieser Stelle ist das Modularisierungsschema redundant, weil es mehrere Möglichkeiten eines sinnvollen Modularisierung gibt, die zum gleichen Ergebnis führt.

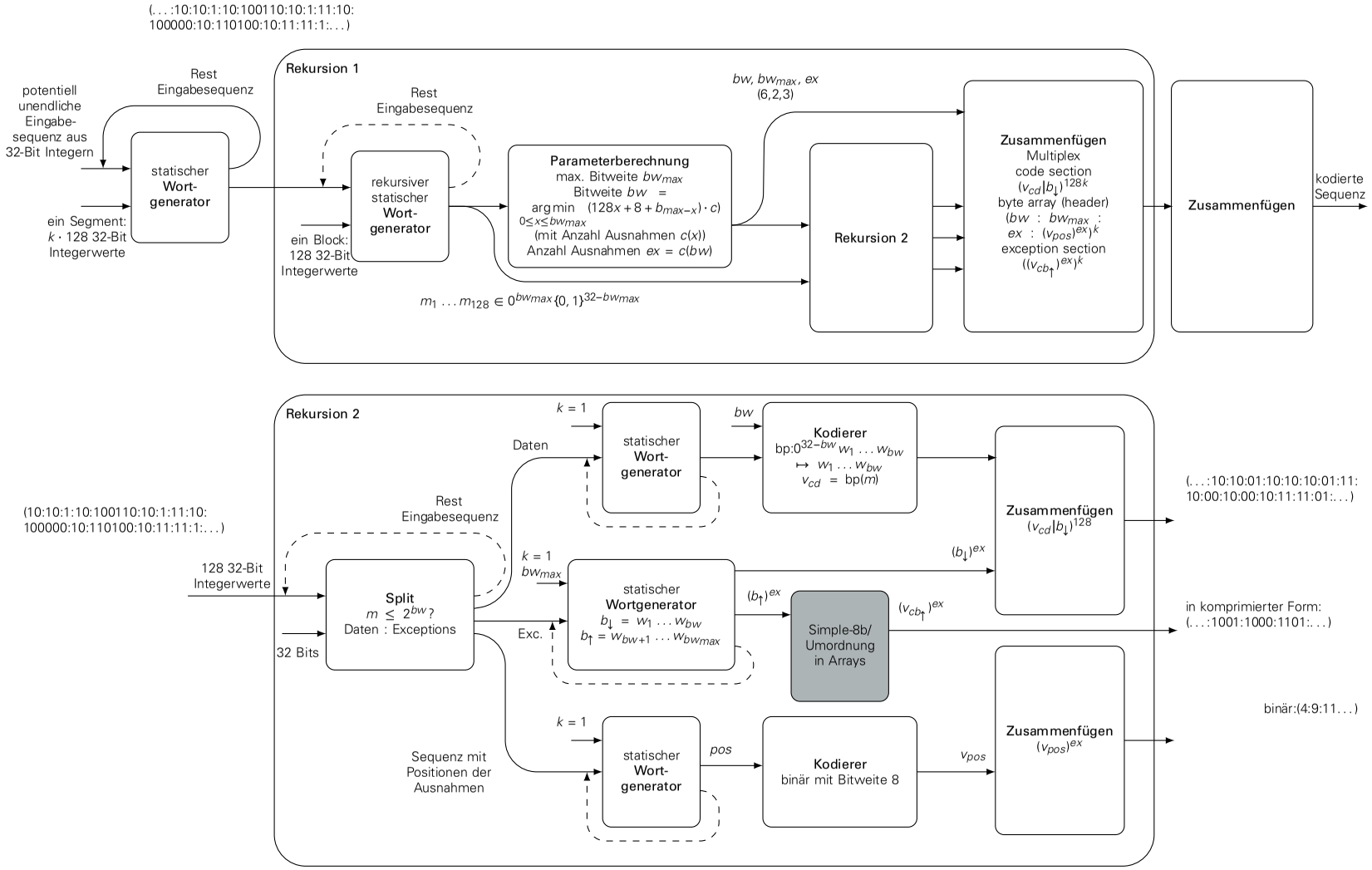


Abbildung 4.12: Kompressionsschema SimplePFOR und FastPFOR

4.4.4 ANMERKUNGEN ZU DIFFERENZKODIERTEN DATEN

[Hé05] beschreibt neben PFOR den Algorithmus PForDelta, welcher sich von erstgenanntem dadurch unterscheidet, dass die einzelnen Werte sortiert vorliegen und differenzkodiert werden. Die Modularisierung mit dem Kompressionsschema gelingt dadurch, dass das gesamte Schema zweimal aufzurufen wird, einmal zur Differenzkodierung und ein zweites Mal das PFOR-Schema nach Abb. 4.7. Selbiges betrifft die Algorithmen NewPFD, OptPFD, SimplePFOR und FastPFOR. Der Einfachheit halber werden sortiert vorliegende Daten als bereits differenzkodiert betrachtet.

4.5 SIMPLE-ALGORITHMEN

Die Algorithmen der Simple-Familie verfolgen den Grundgedanken mit 32 bzw. 64 Bits möglichst viele Integerwerte zu kodieren. Dazu gibt es zu Beginn einen Header aus 4 bzw. 2 Bits, welcher den Kodierungsmodus festlegt. Die restlichen Bits werden mit Daten gefüllt. Teilweise sind Paddingbits vonnöten. In den Modularisierungsschemata für diese Algorithmenfamilie ist das Muster der Symbolunterdrückung mit Lauflängenkodierung vorhanden. Die Algorithmen Simple-9 (S9, Simple-4b), Relative-10 (wegen Zitierfehlern in Literaturrecherchen verschiedener Veröffentlichungen auch Relate-10 genannt) und Carryover-12 (Carry-4b) werden in [AM05] erläutert.

4.5.1 SIMPLE-9

Der erste, einfachste und klarste Algorithmus ist Simple-9. Simple-9 kodiert Gruppen von Integerwerten mit jeweils 32 Bits. Die Anzahl der Elemente der Gruppe hängt von der Größe der einzelnen Zahlen ab. Die 32 Bits werden in einen Header und eine Sektion mit Datenbits aufgeteilt. Der Header besteht aus 4 Deskriptorbits, da 9 Kodierungsmodi existieren (siehe Tab. 4.2). Damit verbleiben 28 Datenbits für eine variable Anzahl von Integerwerten, die mit der gleichen Bitweite binär gepackt werden sollen. Beispielsweise ist die Kodierung von drei Integerwerten mit einer Bitweite $bw = 9$ mit dem Selektor $s = 6$ ein möglicher Kodierungsmodus. Die drei Integerwerte werden mit den 4 Deskriptorbits 0110, $3 \cdot 9 = 27$ Bits, für die kodierten Werte und einem Paddingbit 0, also mit insgesamt 32 Bits gespeichert.

Abbildung 4.13 zeigt das Kompressionsschema für Simple-9. Der Wortgenerator bestimmt die Anzahl k der gemeinsam zu kodierenden Werte und damit im Prinzip auch den Kodierungsmodus bzw. die gemeinsame Bitweite. Wortgenerator und Parameterberechnung haben hier eine so hohe Modulkopplung, dass eine Trennung nicht sinnvoll ist. Die Bitweite, mit der die ersten k Werte $m_1, \ldots, m_k$ vom Beginn der Eingabesequenz mit maximal 28 Bits kodiert werden können, muss kleiner sein als $\lfloor \frac{28}{k} \rfloor$ oder genauso groß. Das ist eine Bedingung, die der erste Wortgenerator als Eingabe benötigt. Die gemeinsame Bitweite bw für k Werte soll nicht größer

Tabelle 4.2: Kodierungsmodi für Simple-9 (aus [AM05])

Selektor	Anzahl Werte	Bitweite *bw*	Anzahl ungenutzte Bits
0	28	1	0
1	14	2	0
2	9	3	1
3	7	4	0
4	5	5	3
5	4	7	0
6	3	9	1
7	2	14	0
8	1	28	0

als notwendig gewählt werden. Deswegen sollen die ersten k Werte mit insgesamt $k \cdot bw \leq 28$ Bits kodiert werden können, für die ersten $k+1$ Werte sollen nicht mehr mit der kleinstmöglichen gemeinsamen Bitweite und maximal 28 Bits kodiert werden können. In der Rekursion werden die einzelnen Werte mit der Bitweite *bw* binär gepackt und am Ende mit dem binär kodierten Kodierungsmodus als gemeinsamen Deskriptor zusammengefügt. Das gegebene Kodierungsschema, die Anzahl der zur Verfügung stehenden Bits und die Größe des Headers sind im Grunde genommen nur austauschbare Parameter. Die Zuordnung zwischen Bitweite und Kodierungsmodus kann man als Wörterbuchkompression betrachten.

4.5.2 SIMPLE-16

Simple-16 [ZLS08] ist eine Erweiterung von Simple-9 mit dem Gedanken, dass mit vier Deskriptorbits 16 und nicht nur 9 Kodierungsmodi möglich sind, dann allerdings mit verschiedenen Bitweiten innerhalb einer 32-Bit-Einheit. Beispielsweise könnte es einen Kodierungsmodus geben, mit dem drei Integerwerte mit 6 Bits und zwei Integerwerte mit 5 Bits kodiert werden. Damit wird in zweierlei Hinsicht Platz gespart. Zum einen dadurch, dass alle möglichen Kodierungsmodi definiert sind und genutzt werden, zum anderen werden unter den Datenbits keine ungenutzt verschenkt. Die Modularisierung gelingt am besten dreistufig durch Einführung einer Zwischenstufe, in welcher die Integerwerte, die in einer 32-Bit-Einheit kodiert werden sollen, in endliche Sequenzen unterteilt werden, die jeweils mit gleicher Bitweite kodiert werden.

Das Kompressionsschema ist in Abbildung 4.14 dargestellt. Der Wortgenerator trifft auf Grundlage des Kodierungsschemas die Entscheidung, wieviele Werte (und auch mit welchen Bitweiten) mit 32 Bits kodiert werden können. An dieser Stelle ist eine Trennung zwischen Wortgenerator und Parameterberechnung genauso wenig sinnvoll wie bei Simple-9. Um dem Kompressionsschema zu entsprechen, sind beide Module dennoch getrennt. Die Parameterberechnung erhält als Eingabeparameter 16 mögliche Fälle, die beschreiben, mit welcher Bitweite bw_i alle Teilsequenzen aus c_i Werten kodiert werden, so dass im Idealfall keine Paddingbits notwendig sind.

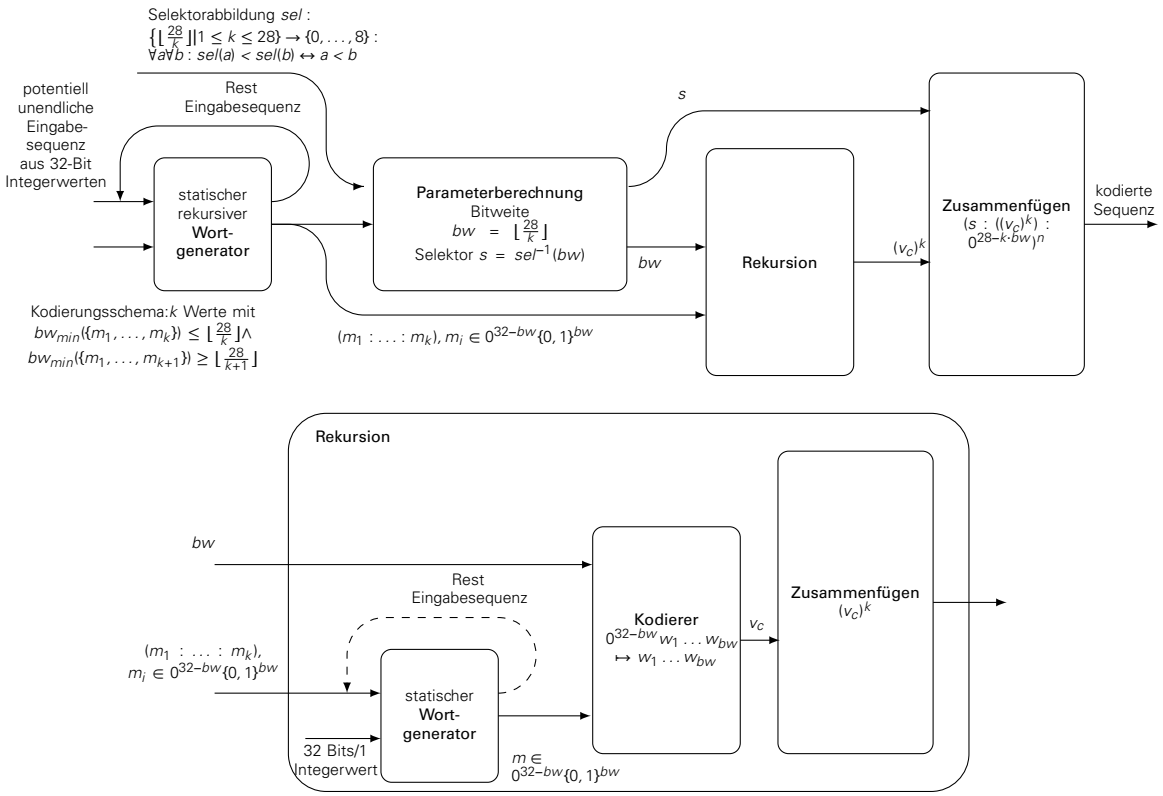


Abbildung 4.13: Kompressionsschema Simple-9

Weiterhin gibt es eine eineindeutige Zuordnung aus den 16 definierten Fällen zum Kodierungsmodus, welcher mit 4 Bits kodiert wird. Die Parameterberechnung legt den Kodierungsmodus und damit die Untergliederung in Teilsequenzen sowie die Bitweiten für die Kodierung selbiger fest. In der Rekursion erhält der Wortgenerator als Eingabe die zu zerlegende endliche Sequenz ($m_1 : \ldots : m_k$) sowie den oben berechneten Parameter ($c_1, \ldots, c_n$), welcher für die Zerlegung benötigt wird. Die Parameterberechnung benötigt keine Teilsequenzen als Eingabe. Sie erhält die Liste der korrespondierenden Bitweiten als Parameter und gibt pro Teilsequenz eine davon aus. Rekursion 2 entspricht der Rekursion beim Binary Packing.

In [YDS09] wird mit S16-128 ein Algorithmus vorgestellt, der sich von Simple-16 nur insofern unterscheidet, dass mehr Kodierungsmodi für kleinere Werte zur Verfügung stehen und weniger für große, um für kleinere Zahlen eine leicht bessere Kompressionsrate zu erzielen. Somit unterscheiden sich die Kompressionsschemen für beide Algorithmen nur in einem eingehenden Parameter.

4.5.3 RELATIVE-10 UND CARRYOVER-12

Relative-10 ähnelt sehr Simple-9, hat aber 10 Kodierungsmodi, bei denen alle Werte mit der gleichen Bitweite kodiert werden (siehe Tab. 4.3). In Abhängigkeit des Kodierungsmodus der vorangegangenen Sequenz sind für die aktuelle Sequenz vier der 10 Kodierungsmodi erlaubt und stehen zur Auswahl. Die Wahl des Deskriptors besteht aus nur 2 Datenbits, da nur eine relative Angabe notwendig ist. Tab. 4.4 ist hierfür die Transfertabelle. Es stehen 30 Datenbits zur Verfügung. Ist zum Beispiel eine Sequenz ($1^{39}) : 5 : \ldots$) gegeben, so werden zunächst mit Kodierungsmodus 1 30 Einsen mit jeweils einem Bit kodiert. Nach Tab. 4.4 sind für die folgende Teilsequenz die Kodierungsmodi 1, 2, 3 und 10 möglich. Sinnvollerweise fällt die Wahl auf Kodierungsmodus 3, so dass die folgenden 9 Einsen und die eine Fünf mit jeweils 3 Bits kodiert werden. Dass die Wahl auf den dritten der vier zur Auswahl stehenden Kodierungsmodi gefallen ist, wird mit dem Selektor 10 als Deskriptor kodiert.

Das Interessante am Kompressionsschema dieses Algorithmus ist, dass die Adaptivität nicht nur die Parameterberechnung, sondern auch den Wortgenerator beeinflusst. Parameterberechnung und Wortgenerator weisen eine ausgesprochen hohe Modulkopplung auf (siehe Abb. 4.15), eine Trennung ist wie bei allen Simple-Algorithmen nicht sinnvoll. Der Wortgenerator muss das Kodierungsschema und die Transfertabelle kennen. Er entscheidet die Länge k der ersten Teilsequenz (und damit im Prinzip auch, ob sie mit Kodierungsmodus 1, 2, 3 oder 10 kodiert wird). Die adaptive Parameterberechnung erhält als Startdefinition die Information, dass die Kodierungsmodi 1, 2, 3 und 10 zur Auswahl stehen. Der passende Kodierungsmodus wird ausgewählt und mit der Transfertabelle der (relative) Selektor festgelegt, welcher beim Zusammenfügen als Deskriptor dient. Außerdem gibt die Parameterberechnung den Kodierungsmodus aus, der wiederum dem Wortgenerator zur Festlegung der nächsten Teilsequenz und der Parameterberechnung zur Festlegung des Selektors als Ein-

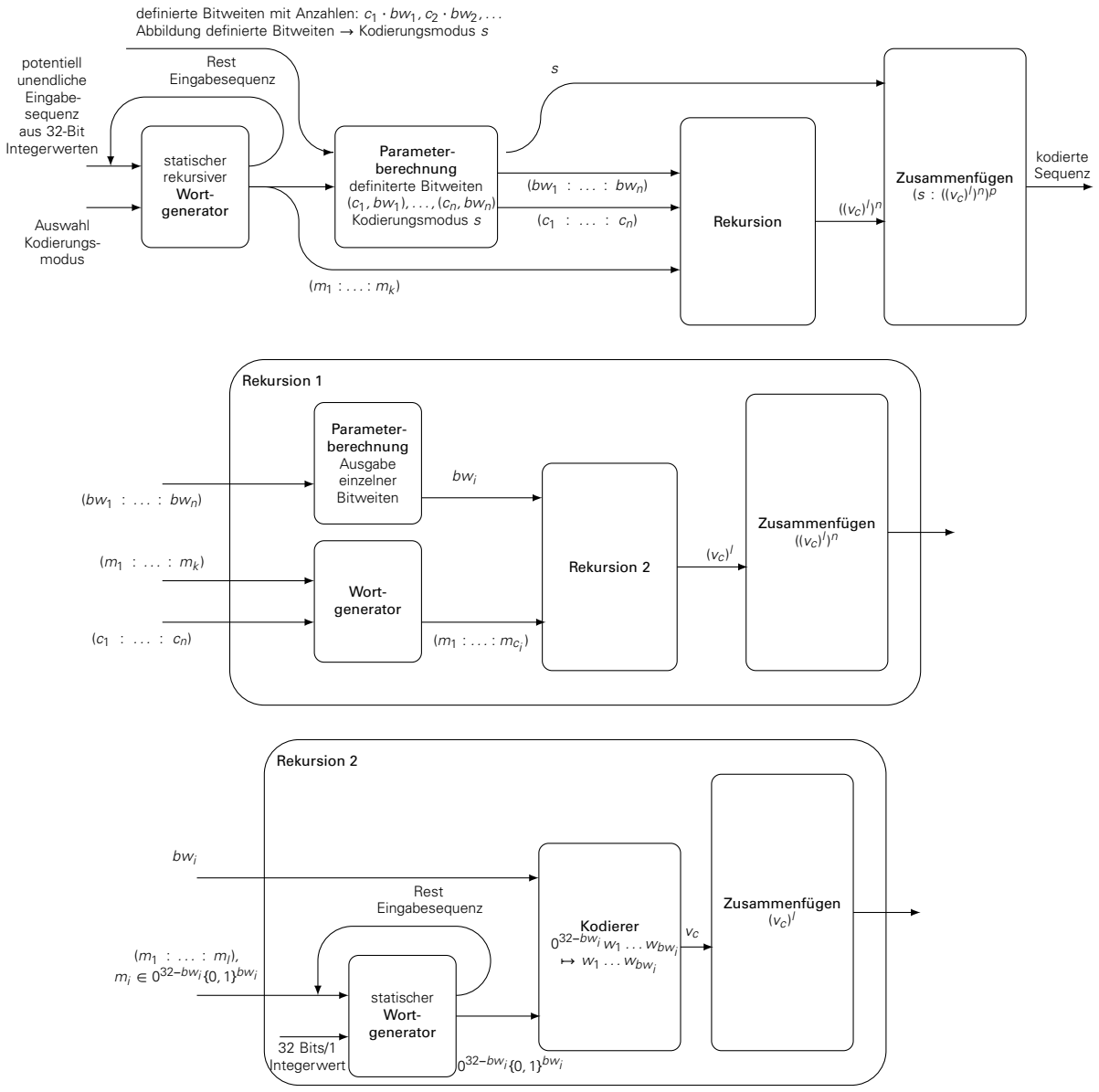


Abbildung 4.14: Kompressionsschema Simple-16 und S16-128

Tabelle 4.3: Kodierungsmodi für Relative-10 (aus [AM05])

Kodierungsmodus	Anzahl Werte	Bitweite *bw*	Anzahl ungenutzte Bits
1	30	1	0
2	15	2	0
3	10	3	0
4	7	4	0
5	6	5	2
6	5	6	0
7	4	7	2
8	3	10	0
9	2	15	0
10	1	30	0

Tabelle 4.4: Kodierung des nächsten Kodierungsmodus in Abhängigkeit vom aktuellen Kodierungsmodus bei Relative-10 (aus [AM05])

Aktueller	möglicher nächster Kodierungsmodus									
Kodierungsmodus	1	2	3	4	5	6	7	8	9	10
1	00	01	10							11
2	00	01	10							11
3		00	01	10						11
4			00	01	10					11
5				00	01	10				11
6					00	01	10			11
7						00	01	10		11
8							00	01	10	11
9							00	01	10	11
10							00	01	10	11

gabe dient. Die Rekursion gleicht der des Algorithmus Simple-9.

Carryover-12 ist eine in [AM05] beschriebene Erweiterung des Algorithmus Relative-10 und folgt dem gleichen Kompressionsschema. Ein Selektor besteht aus 2 Bits. Relative-10 lässt bei den Kodierungsmodi 5 und 7 (siehe Tab. 4.3) zwei Bits ungenutzt, welche bei Carryover-12 für den Selektor der nächsten zu kodierenden Einheit verwendet werden. Für die nächste Einheit stehen dann 32 Datenbits zur Verfügung. Sollten nur 30 Datenbits benötigt werden, können wiederum die verbleibenden zwei Bits für den nächste Selektor genutzt werden. Der einzige Unterschied im Kompressionsschema zu Relative-10 besteht darin, dass die Parameterberechnung die Information, ob 30 oder 32 Datenbits für die nächste Sequenz zur Verfügung stehen, ausgeben muss, weil der Wortgenerator diese Information benötigt. Weiterhin sind beim Zusammenfügen keine Paddingbits nötig.

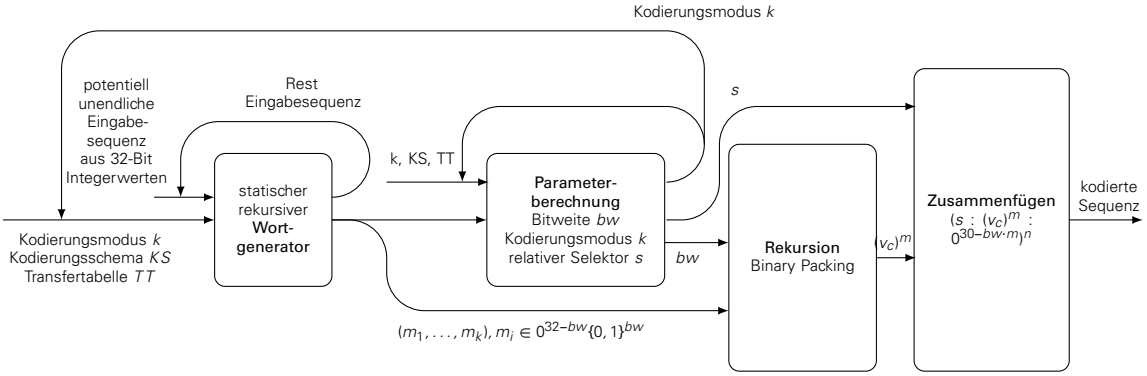


Abbildung 4.15: Kompressionsschema Relative-10

4.6 BYTEORIENTIERTE KODIERUNGEN

Byteorientierte Kodierungen bilden 32-Bit-Integerwerte, also einen Blockcode, auf einen Code mit meist variabler Länge ab, indem eine bestimmte Anzahl führender Nullen und damit nicht signifikanter Bits entfernt und stattdessen ein Längendeskriptor eingeführt wird, der die Anzahl der Bytes, die für die Darstellung eines Integerwertes benötigt werden, angibt. Byteorientierte Kodierungen nutzen damit das RLE- und das Symbolunterdrückungsmuster. Genauer definiert wird der grundlegende Gedanke in [SGR+11]:

Definition 4.6.1 (*Byteorientierte Kodierungen*)

Eine Kodierung ist byteorientiert gdw. sie die folgenden Bedingungen erfüllt.

1. *Alle signifikanten Bits der binären Repräsentation bleiben erhalten.*
2. *Jedes Byte enthält nur Bits eines einzigen Integerwertes.*
3. *Die Datenbits eines einzelnen kodierten Bytes behalten die gleiche Reihenfolge, die sie im originalen Integerwert haben.*
4. *Alle Bits eines einzelnen Integerwertes gehen den Bits des folgenden Integerwertes voraus.*

Eine byteorientierte Kodierung hat feste Länge, wenn jeder Integerwert mit der gleichen Zahl an Bytes kodiert wird. Anderenfalls ist sie von variabler Länge.

Weiterhin klassifiziert [SGR+11] bereits etabilerte und neu eingeführte Algorithmen nach der Kodierung des Längendeskriptors (*U* für *unär* bzw. *b* für *binär*) und der Anordnung von Deskriptoren und Daten (*S* für *split*, *P* für *packed* und *G* für *group*) und benennt alle, auch bereits bestehende, Formate nach diesem Schema (siehe Tab. 4.5). Diese Formate sowie RLE VByte werden im folgenden beschrieben und modularisiert.

Tabelle 4.5: Varint-Formate

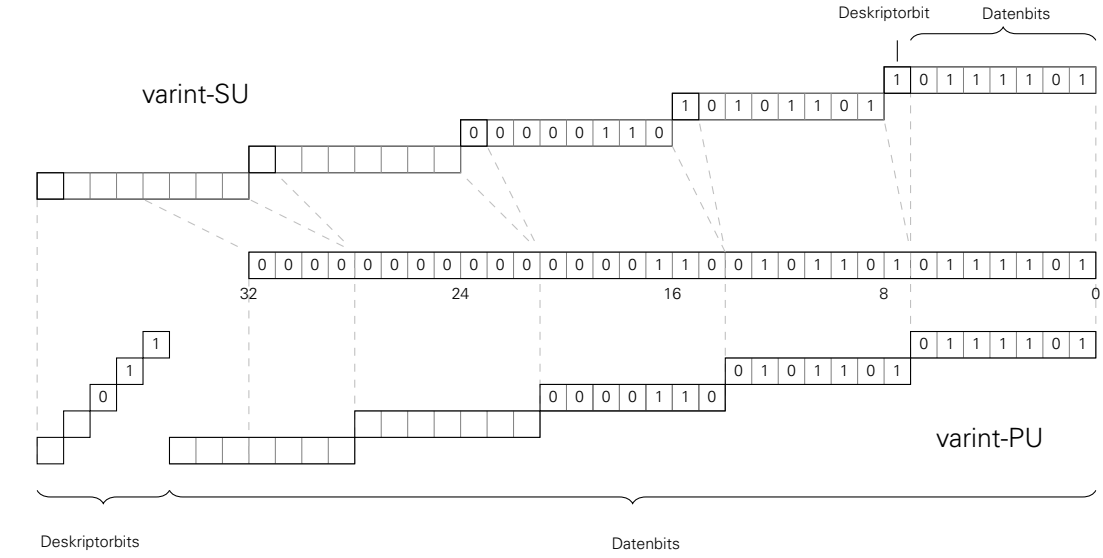
		Deskriptorarrangement		
		split	packed	group
Kodierung Deskriptor	unär	Varint-SU (Kap. 4.6.1)	Varint-PU (Kap. 4.6.1)	Varint-GU (Kap. 4.6.2)
	binär	-	Varint-PB (Kap. 4.6.3)	Varint-GB (Kap. 4.6.4)

4.6.1 VARINT-SU UND VARINT-PU

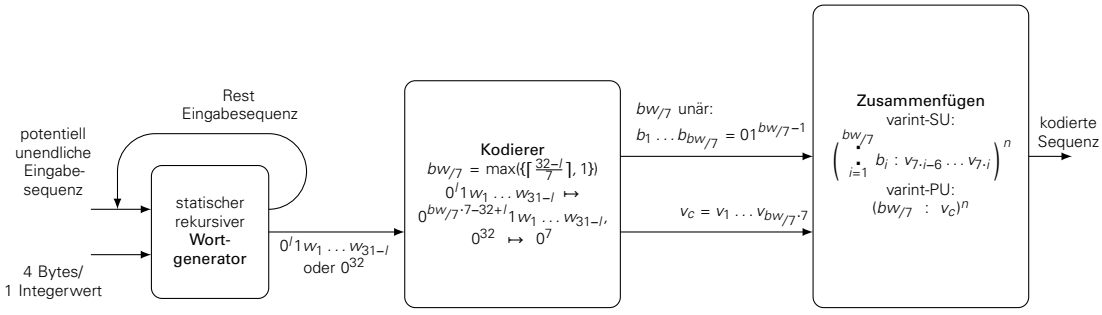
Die ältesten Referenzen zu VByte (Varint-SU) stammen aus den 1970er und 1980er Jahren (siehe Abb. 4.1). VByte kodiert 32-Bit-Integerwerte mit ein bis 5 Bytes, wobei ein Byte aus einem Deskriptorbit und 7 Datenbits besteht. Ebenso ist dies bei Varint-PB der Fall, beide Algorithmen unterscheiden sich nur in der Anordnung der Daten- und Deskriptorbits. Während bei VByte ein Bit pro Byte als Deskriptor dient, steht bei Varint-PB der gesamte Deskriptor an einem Ende des komprimierten Integerwertes. Ein Beispiel zeigt Abbildung 4.16a. Nicht belegte Bits bedeuten, dass diese im Beispiel nicht benötigt und weggelassen werden. Der Integerwert wird in komprimierter Form mit 3 statt 4 Bytes kodiert. Beide Formate haben den gleichen modularen Aufbau (siehe Abb. 4.16b). Der rekursive statische Wortgenerator gibt immer eine Zahl aus. Da die Kodierung der Eingabe bei diesen Algorithmen soweit spezifiziert ist, dass die Zahlen als 32-Bit-Integerwerte kodiert sind, ist die ausgegebene Zahl ein eingebetteter Lauf von der Form $0^l 1 w_1 \ldots w_{31-l}$ (bzw. 0^{32} für den Wert 0). Der Deskriptor $bw_{/7}$ gibt die Anzahl der für die Datenbits benötigten 7-Bit-Einheiten an. Allein aus dem komprimierten Wert ohne Deskriptor ist die Lauflänge der bei der Kodierung unterschlagenen Nullen nicht ermittelbar, schon weil eine Folge von Werten nicht mehr dekodierbar ist. Die Lauflänge ist allein aus dem Deskriptor (und dem Wissen, dass es sich um 32-Bit-Integerwerte handelt) ersichtlich. Somit ist das Lauflängenmuster bei Varint-SU und Varint-PU begründbar. Da das Zeichen 0 eine Sonderstellung einnimmt, weil führende Nullen als Lauf betrachtet werden, findet sich hier, wie bei allen Varint- Algorithmen, auch eine Symbolunterdrückung. Beide Algorithmen sind statische Wörterbuch-Komprimierungen und unterscheiden sich im Kompressionsschema nur im Modul des Zusammenfügens. Das Symbol : wird hier als Konkatenationssymbol für abzählbar viele Werte verwendet.

4.6.2 VARINT-GU

Bei den gruppierenden Varint-Algorithmen werden die Deskriptoren für mehrere Werte in einem separaten Byte vorangestellt. Der unäre Deskriptor $bw_{/8}$ gibt für einen Integerwert an, wie viele 8-Bit-Einheiten für die Darstellung der Daten benötigt werden. Dementsprechend haben die Deskriptoren eine Länge von einem bis vier Bits.

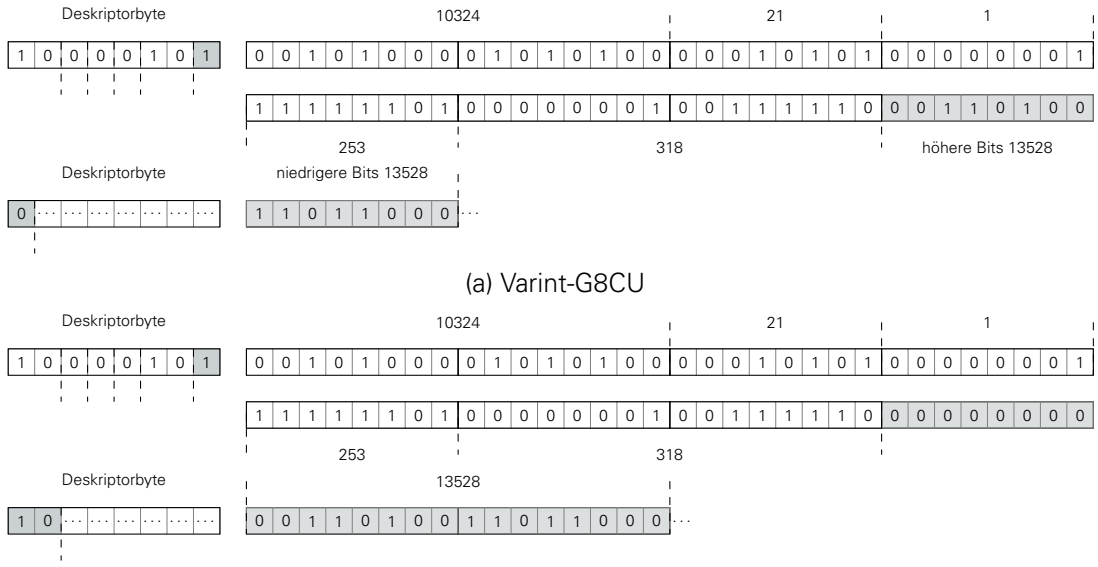


(a) Arrangement des Deskriptors



(b) Kompressionsschema

Abbildung 4.16: Varint-SU und Varint-PU



(a) Varint-G8CU

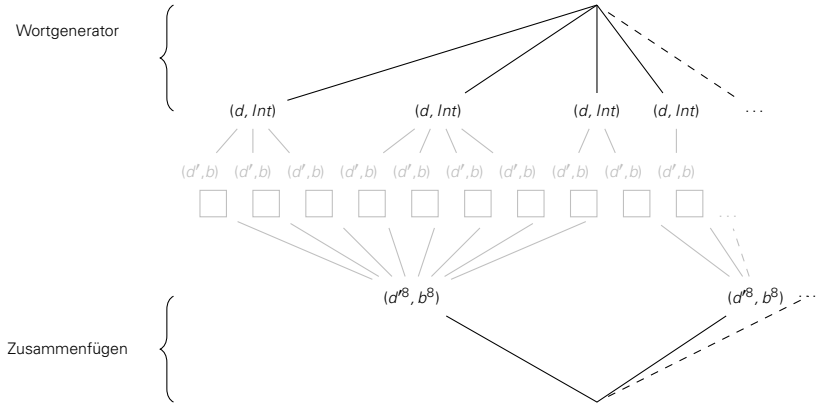
(b) Varint-G8IU

Abbildung 4.17: Beispiel Varint-GU

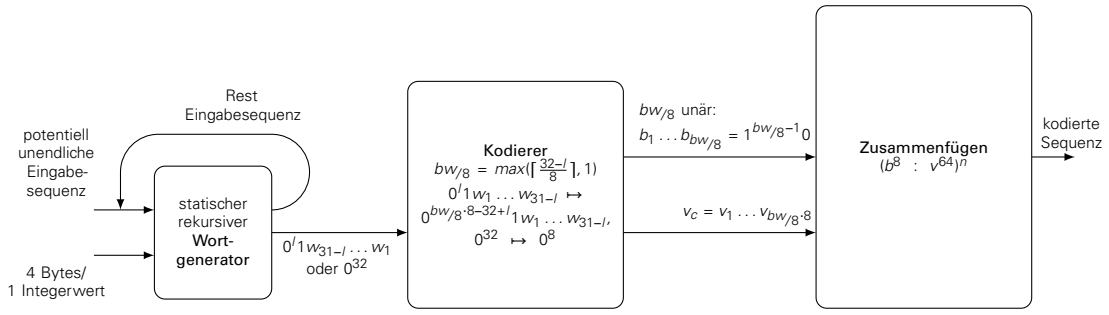
Zusammengefügte Einheiten bestehen aus einem Deskriptorbyte und 8 Datenbytes, also insgesamt 9 Bytes. Im Normalfall ist davon auszugehen, dass beim Zusammenfügen die 9-Byte-Grenzen überspannt werden, wenn der Platz wie bei Varint-G8CU vollständig (*complete*) genutzt werden soll. Sollen die Werte immer im Ganzen in einer 9-Byte-Einheit enthalten sein, muss der nicht mehr nutzbare Platz mit Nullen aufgefüllt werden (*incomplete*, Varint-G8IU). Sollen beispielsweise die Zahlen 10324, 21, 1, 253, 318 und 13528 kodiert werden, werden ohne Deskriptoren 9 Bytes benötigt, da 10324 mit zwei Bytes, 21, 1 sowie 253 mit einem Byte und 318 sowie 13528 mit zwei Bytes kodiert werden.

Mit Varint-G8CU werden die 8 Datenbytes vollständig genutzt, die kodierte Form des Wertes 13528 überspannt die 9-Byte-Grenze (siehe Abb. 4.17a). Soll diese Grenze nicht überspannt werden, muss der Wert 13528 vollständig in die folgenden 9 Bytes verschoben und der ungenutzte Platz mit Paddingbits gefüllt werden (siehe Abb. 4.17b). Deskriptorbits sind folgendermaßen zu verstehen. Jedem Datenbyte ist ein Deskriptorbit *d*' zugeordnet. Der Wert 1 sagt aus, dass die Zahl mit dem zugeordneten Byte noch nicht endet, eine 0 bedeutet, dass im nächsten Byte eine neue Zahl beginnt. Paddingbits für das Deskriptorbyte im Algorithmus Varint-G8CU sind deshalb immer Einsen.

Die Modularisierung bei beider Algorithmen (siehe Abb. 4.18b und Abb. 4.19b) ähneln der von Varint-SU und Varint-PU nur insofern, dass der gleiche Wortgenerator genutzt wird. Der Kodierer gibt neben dem komprimierten Wert, dessen Länge in diesem Falle kein Vielfaches von 7, sondern von 8 Bits ist, eine unäre Längenangabe aus, die sich auf 8 Bits bezieht. Bei vielen Algorithmen stimmt die hierarchische Struktur beim Unterteilen und Zusammenfügen der Datensequenzen überein. Wie



(a) Hierarchische Datenorganisation bei Varint-G8CU



(b) Kompressionsschema Varint-G8CU

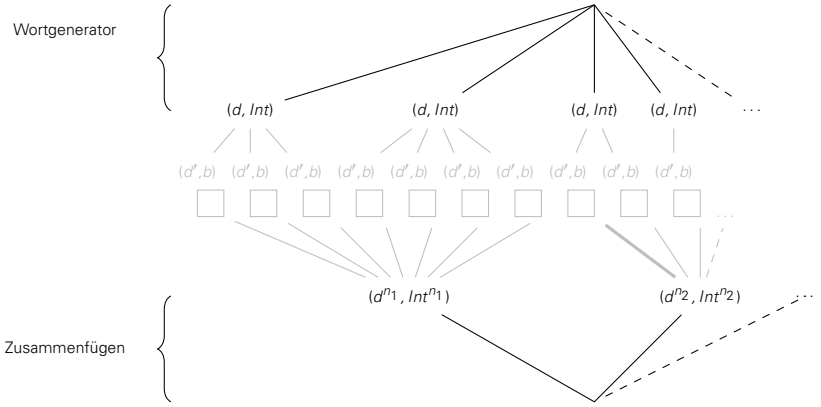
Abbildung 4.18: Varint-G8CU

in Abb. 4.18a zu sehen, ist dies bei Varint-G8CU nicht der Fall. Die Ausgaben des Kodierers können deswegen besser als potentiell unendliche Sequenzen von Bits bzw. Bytes gesehen werden, die statisch und damit datenunabhängig zusammengefügt werden. Die untere, grau dargestellte Ebene ist im Kompressionsschema nicht dargestellt.

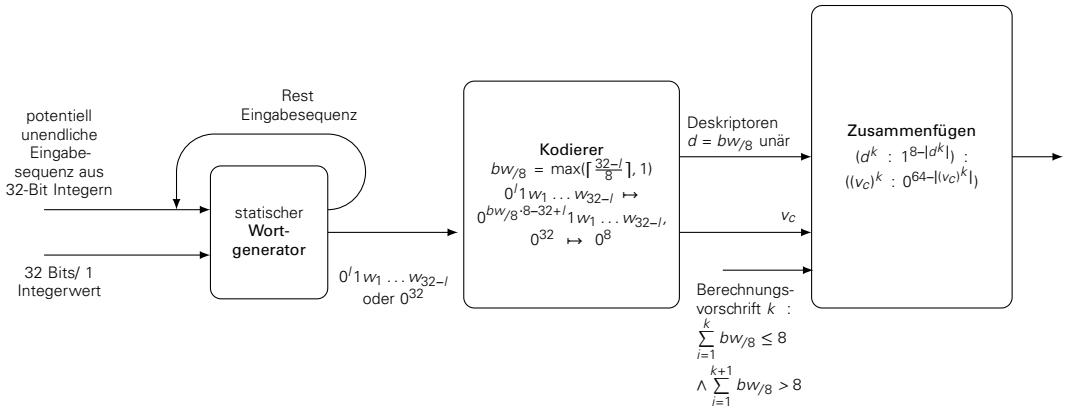
Die Datenhierarchie von Varint-G8IU (siehe Abb. 4.19a) gleicht der von Varint-G8CU, beide Algorithmen unterscheiden sich nur in der Art und Weise wie Daten zusammengefügt werden. Bei Varint-G8IU werden keine 8-Byte-Grenzen überspannt. Stattdessen wird nicht nutzbarer Platz mit Nullen/Einsen aufgefüllt. Varint-G8IU ist der einzige in dieser Arbeit betrachtete Algorithmus, welcher komprimierte Daten semiadaptiv zusammenfügt. Ebenso wie bei Varint-G8CU ist auch bei Varint-G8IU die unterste graue Datenorganisationsebene nicht im Kompressionsschema dargestellt.

4.6.3 VARINT-PB

Auch Varint-PB [Gro95], [Dea09] ist einer der nicht neu erdachten Algorithmen. Da Integerwerte $< 2^{31}$ ohne führende Nullen nur $n \cdot 8 - 2$ Bits mit $1 \leq n \leq 4$ benötigen, kann der Längendeskriptor binär mit nur 2 Bits ausgedrückt und den Datenbits vorangestellt werden. Abbildung 4.20a zeigt ein Beispiel für die Kodierung des Wertes 104125. Der Kodierer im Kompressionsschema berechnet als Deskriptor die Anzahl

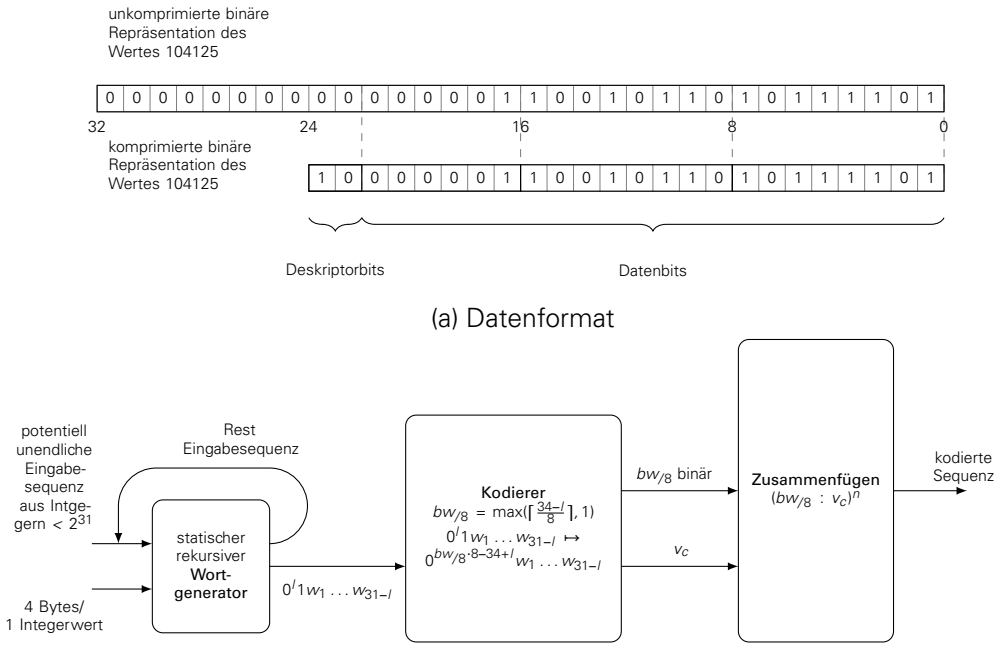


(a) Hierarchische Datenorganisation



(b) Kompressionsschema

Abbildung 4.19: Varint-G8IU



(a) Datenformat

(b) Kompressionsschema

Abbildung 4.20: Varint-PB

der benötigten Bytes und entfernt eine passende Anzahl führender Nullen (siehe Abb. 4.20a). Im Modul des Zusammenfügens wird der binär kodierte Dekriptor mit dem komprimierten Wert konkateniert. Dabei ist in dieser Arbeit die Abbildung der binären Repräsentation der möglichen Deskriptorwerte als {(1, 00), (2, 01), (3, 10), (4, 11)} definiert. Die binäre Repräsantation 10 des Deskriptors in Abbildung 4.20a steht für den Wert 3 und sagt aus, dass die Zahl 104125 mit $3 \cdot 8 - 2$ Bits kodiert wird. Natürlich sind auch anders definierte Längenangaben möglich. Plausibel ist, dass es keinen Algorithmus Varint-SB geben kann, weil der Deskriptor aus 2 Bits nicht auf 1, 3 oder 4 Datenbytes aufgeteilt werden kann.

4.6.4 VARINT-GB

Varint-GB [SGR+11], k-wise Null Suppression [SGL10] oder group varint [Dea09] gruppiert 2-Bit-Längendeskriptoren für jeweils vier aufeinanderfolgende Integerwerte in einem Byte. Die vier Integerwerte werden mit jeweils bis zu vier Bytes kodiert. Damit können im Unterschied zu Varint-PB auch Integerwerte mit den Bitweiten 31 oder 32 kodiert werden. Bei der Modularisierung (siehe Abb. 4.21) werden einzelne Integerwerte kodiert und ihre Deskriptoren berechnet. Zusammengefügt werden jeweils vier Werte und ihre Deskriptoren.

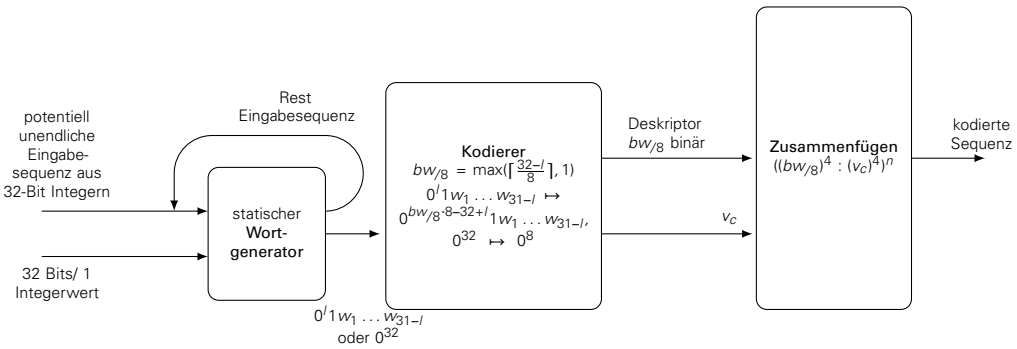


Abbildung 4.21: Kompressionsschema Varint-GB

Tabelle 4.6: Vergleich der Module der Varint-Algorithmen

	Varint-SU	Varint-PU	Varint-G8CU	Varint-G8IU	Varint-PB	Varint-GB
Wortgenerator	statisch (k=1)					
Kodierer	Berechnung		Berechnung		Berechnung	Berechnung
Zusammenfügen	statisch $((d : v)^m)^n$	statisch $(d^m : v^m)^n$	statisch $(d^8 : v^8)^n$	semiadaptiv	statisch $(d : v)^n$	statisch $(d^4 : v^4)^n$

4.6.5 VERGLEICH DER MODULE DER VARINT-ALGORITHMEN

Tabelle 4.6 zeigt den Vergleich der Module aller sechs Varint-Algorithmen. Die sechs Varint-Algorithmen gleichen sich im Wortgenerator, der jeweils einen Integerwert ausgibt. Varint-SU und Varint-PU sowie Varint-G8CU und Varint-G8IU nutzen den gleichen Kodierer und unterscheiden sich nur beim Zusammenfügen. Bei Algorithmen mit unären Deskriptoren steht der Deskriptor d in der Tabelle für ein Deskriptorbit, bei Algorithmen mit binärem Deskriptor für den gesamten Deskriptor pro Integerwert, also für 2 Bits. Varint-G8IU benötigt ein Modul des Zusammenfügens, welches datenabhängig entscheidet, wie viele Werte zusammengefasst werden. bei allen anderen Algorithmen ist das Modul des Zusammenfügens statisch. Für Varint-G8IU gibt es außerdem die Möglichkeit einer zweistufigen Modularisierung, so dass der erste Wortgenerator die Anzahl der gemeinsam zusammenzufügenden Werte semiadaptiv bestimmt und die einzelnen Werte erst in der Rekursion kodiert werden. Es fällt auf, dass das Kompressionsschema noch eine gewisse Redundanz besitzt, in dem Sinne, dass es für Algorithmen mehrere plausible Möglichkeiten zur Modularisierung gibt.

4.6.6 RLE VBYTE

RLE VByte [AGOS13] ist ein auf VByte basierender Algorithmus, der im Kontext der Kompression von invertierten Indices Anwendung findet. Seine Annahme aus dem Kontext ist, dass sich bei der Kodierung von Abständen zwischen Integerwerten in Listen häufig längere Läufe aus Einsen finden, die komprimierend lauflängenkodiert

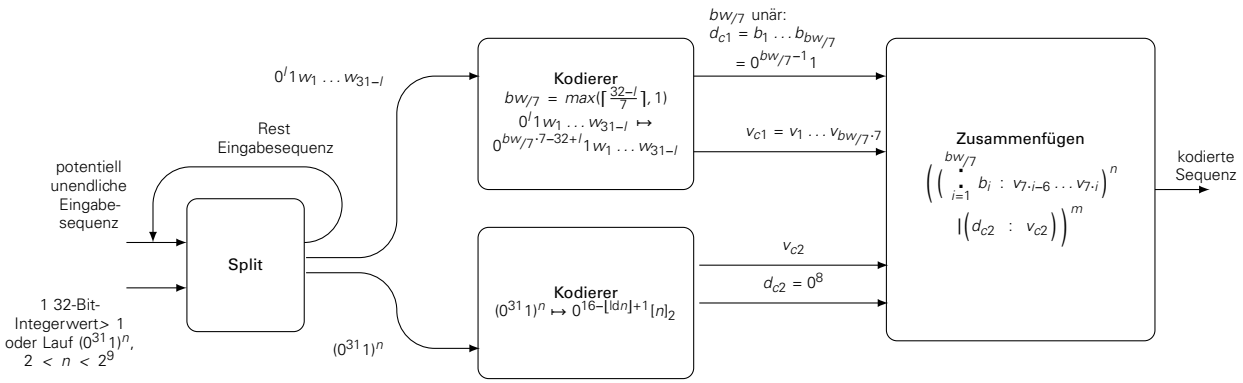


Abbildung 4.22: Kompressionsschema RLE VByte

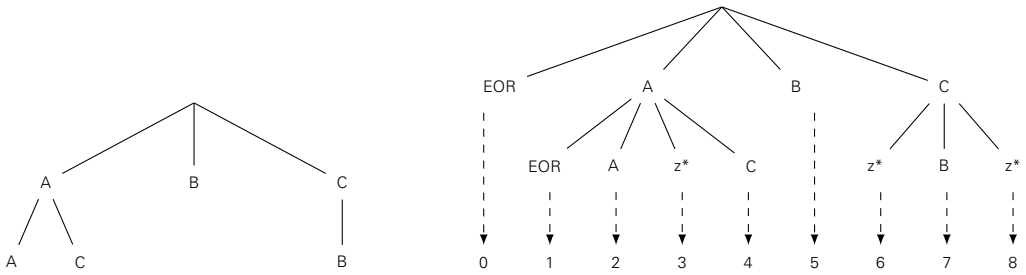
werden sollen. RLE VByte nutzt also in zweierlei Hinsicht das Muster der Symbolunterdrückung durch Lauflängenkodierung. Zum einen werden auf Bitebene führende Nullen unterdrückt, zum anderen oberhalb der Kodierungsebene Läufe von Einsen. Genau wie VByte (siehe Kap. 4.6.1) kodiert RLE VByte Integerwerte mit unärem und auf die einzelnen Datenbytes aufgeteilten Längendeskriptor. Da alle Eingabewerte größer als 0 sind, kann 0^8 als Deskriptor genutzt werden, um einen Lauf aus Einsen anzukündigen, dessen Lauflänge ($< 2^9$) im nächsten Byte kodiert ist. Die Modularisierung (siehe Abb. 4.22) gelingt mit einem Split in Läufe und einzelne Werte und dem Zusammenfügen in der gleichen Reihenfolge.

4.7 WÖRTERBUCHALGORITHMEN

Wörterbuchkompression wurde als Muster bereits in Kapitel 3.5 vorgestellt. Gerade bei der Kodierung von Symbolen ohne metrische Eigenschaften kommt dieses Muster zur Anwendung. Im Folgenden werden zwei Wörterbuchalgorithmen vorgestellt.

4.7.1 ZIL

ZIL ist je nach Anwendung ein statischer oder semiadaptiver Algorithmus für eine Wörterbuchkompression, welcher in [ZIJ93] beschrieben ist. Der Eingabedatenstrom ist eine potentiell unendliche Sequenz aus Symbolen eines Alphabets Σ. Auf dem Alphabet ist eine Ordnung definiert. Weiterhin ist eine Menge zulässiger Zeichenketten definiert, in die die Eingabesequenz zerlegt werden kann. Diese Menge wird entweder semiadaptiv mit einem anderen Algorithmus berechnet oder sie ist aus dem Kontext gegeben. ZIL berechnet aus dieser Menge einen Code, so dass, auch wenn mehrere Symbole zusammen mit einem Codewort kodiert werden, die Ordnung erhalten bleibt. Aus der Menge der zulässigen Zeichenketten lässt sich ein unvollständiger Parsebaum erstellen. Ein Beispiel für das Alphabet $\Sigma = \{A, B, C\}$ und



(a) Unvollständiger Parsebaum (b) Konvertierter Parsebaum

Abbildung 4.23: Parsebäume ZIL

Tabelle 4.7: Beispiele für Kodierungen mit ZIL

Zeichenkette			Code
EOR			0
A:EOR			1
A:A:	...		2:...
A:	B:	...	3:5:...
A:C:	...		4:...
B:	...		5:...
C:	EOR		6:0:...
C:	A:EOR		6:1
C:	A:A:	...	6:2:...
C:	A:	B:...	6:3:5:...
C:	A:C:	...	6:4:...
C:B:	...		7:...

die Menge {*A, AA, AC, B, C, CB*} zeigt Abbildung 4.23a. ZIL konvertiert unvollständige Parsebäume so, dass bei geeigneter Zuordnung der Blattknoten zu Codewörtern die Ordnung zwischen Zeichenketten in komprimierter Form erhalten bleibt. Lücken zwischen Geschwisterknoten, in denen ein oder mehrere Symbole fehlen, werden durch das Zilchsymbol *z**, mit welchem das Alphabet erweitert wird, aufgefüllt. Dieses ist ein einfacher Platzhalter. Ein *EOR*-Symbol wird zum Alphabet hinzugefügt. Dieses ist gemäß der Ordnungsrelation kleiner als alle anderen Symbole und symbolisiert das Ende einer Zeichenkette. Jedem inneren Knoten im Parsebaum wird ein *EOR*-Kindknoten als erstes Kind eingefügt, sofern der erste Kindknoten dem kleinsten Element des Alphabets, im Beispiel dem Symbol *A*, entspricht. Für jeden Kindknoten wird ein Codewort festgelegt. Dabei werden die Codewörter von links nach recht größer (siehe Abb. 4.23b).

Tabelle 4.7 zeigt anhand einiger Beispiele, dass durch die so definierte Abbildung die Ordnung der komprimierten Zeichenketten der der unkomprimierten Zeichenketten entspricht. Das Mapping *wb* für den Kodierer wird in der Parameterberechnung erstellt. Im eher unspektakulären Kompressionsschema für die statische Variante (siehe Abb. 4.24) hat der Wortgenerator die Aufgabe, die Eingabezeichenkette richtig zu trennen. Beginnt diese beispielsweise mit (A:C:...), so gibt es keine Möglichkeit beide getrennt zu kodieren. Um den interessantesten Aspekt des Algorithmus, die

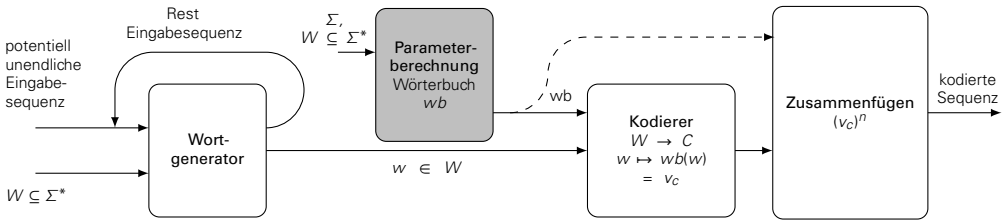


Abbildung 4.24: Statisches Kompressionsschema ZIL

Berechnung des Wörterbuches, kümmert sich die Parameterberechnung. Das Wörterbuch oder zumindest die Menge W muss mit gespeichert werden, wenn nicht vorausgesetzt werden kann, dass der Dekodierer ohne diese Information dekodieren kann. Eben diesem Kompressionsschema folgen Algorithmen, die sich mit ähnlichen Aspekten der Stringkompression beschäftigen, wie zum Beispiel ALM ([ALM96]).

4.7.2 SIGMAKODIERTE INVERTIERTE DATEIEN

Sigmakodierung [TS07] ist ein Wörterbuchalgorithmus zur Kodierung von Integerwerten, welcher ihre metrischen Eigenschaften im Kodierer nicht ausnutzt. Kompression findet auf zweierlei verschiedene Weise statt. Die Erstellung und Nutzung eines Wörterbuches, welches den zu kodierenden Werte aufgrund ihrer Häufigkeit verschieden große natürliche Zahlen zuordnet, ist der erste Ansatzpunkt zur Kompression. Hauptaugenmerk liegt aber auf der Kompression des erstellten Wörterbuches, bei welcher Differenzkodierung und Carryover-12 von Bedeutung sind, so dass auch Teile der Parameterberechnung mit dem Komprimierungsschema ausgedrückt werden können.

Zunächst wird für eine endliche Sequenz $(w_1 : \ldots : w_k)$ ein Wörterbuchprototyp berechnet. Jedem auftauchenden Wert $w \in W = \bigcup_{i=1}^{k} w_i$ wird dabei seine Häufigkeit $c(w)$ zugewiesen. Dabei werden die verschiedenen Werte aus W nach der Ordnung ihrer Häufigkeiten c lückenlos auf natürliche Zahlen abgebildet (sigmakodiert). Häufiger auftauchenden Werten wird damit eine kleinere Zahl zugewiesen. Das Format der komprimierten Daten soll am Ende das Dekrement der Anzahl der Wörterbucheinträge, die Liste der verschiedenen Einträge, geordnet nach der komprimierten Form sowie die komprimierten Daten an sich enthalten. Beispielsweise soll die Sequenz (5:3:12:5:3:5:5:3:12:4) durch 3:(5:3:12:4):(0:1:2:0:1:0:0:1:2:3) ersetzt werden. Das Wörterbuch hat 4 Einträge, es bildet 5 auf 0, 3 auf 1, 12 auf 2 und 4 auf 3 ab. Was die Speicherung des Wörterbuches betrifft, kommt noch eine Optimierung zum Tragen. Da zur Kompression auf Bitebene Binary Packing genutzt werden soll, ist für Werte, deren komprimierte Formen sich durch gleiche Bitweiten auszeichnen, die Abbildung von unkomprimierten auf komprimierten Wert austauschbar, ohne dass sich am Kompressionsverhältnis etwas ändert. Ob der Wert 12 binär als 10 und der Wert 4 als 11 kodiert wird oder 12 als 11 und 4 als 10, macht keinen Unterschied. So werden innerhalb des Wörterbuches Werte, deren komprimierte Formen die gleichen Bitweiten haben (im Beispiel 5 und 3 sowie 12 und 4), aufsteigend geordnet ((3:5):(4:12)).

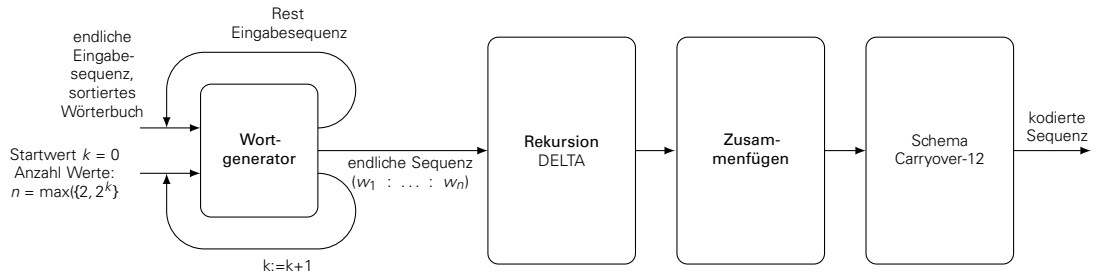
Die Abbildung mit Umordnung ist im Kompressionsschema mit $sigma_{opt}(w)$ bezeichnet. Die Werte mit gleicher Bitweite werden differenzkodiert ((3:1):(4:7)). Es ergibt sich eine Ersetzung der Eingabesequenz durch 3:(3:1:4:7):(1:0:3:1:0:1:1:0:3:2). Das gesamte Wörterbuch sowie die kodierte Sequenz werden anschließend mit Carryover-12 binär gepackt.

Den Algorithmus, welcher das Wörterbuch komprimiert, zeigt Abbildung 4.25a. Er erhält eine endliche Sequenz von Integerwerten als Eingabe. In dieser Sequenz liegen nacheinander 2^1, 2^1, 2^2, 2^3,... Werte aufsteigend sortiert vor. Da nacheinander 2^1, 2^1, 2^2, 2^3,... Werte differenzkodiert werden, wird dafür ein zweistufiges Schema mit einem adaptiven ersten Wortgenerator benötigt. Der Algorithmus Carryover-12 weist eine ganz andere Datenordnung auf (siehe Abb. 4.26), so dass beide Algorithmen nicht ineinander verschachtelt miteinander verbunden werden können. Das Kompressionsschema muss für die endliche Sequenz, ähnlich wie bei den PFOR-Algorithmen mit vorgeschalteter Differenzkodierung, deshalb zweimal hintereinander aufgerufen werden. Eine allgemeine Begründung hierfür lieferte Abschnitt 2.3.3.

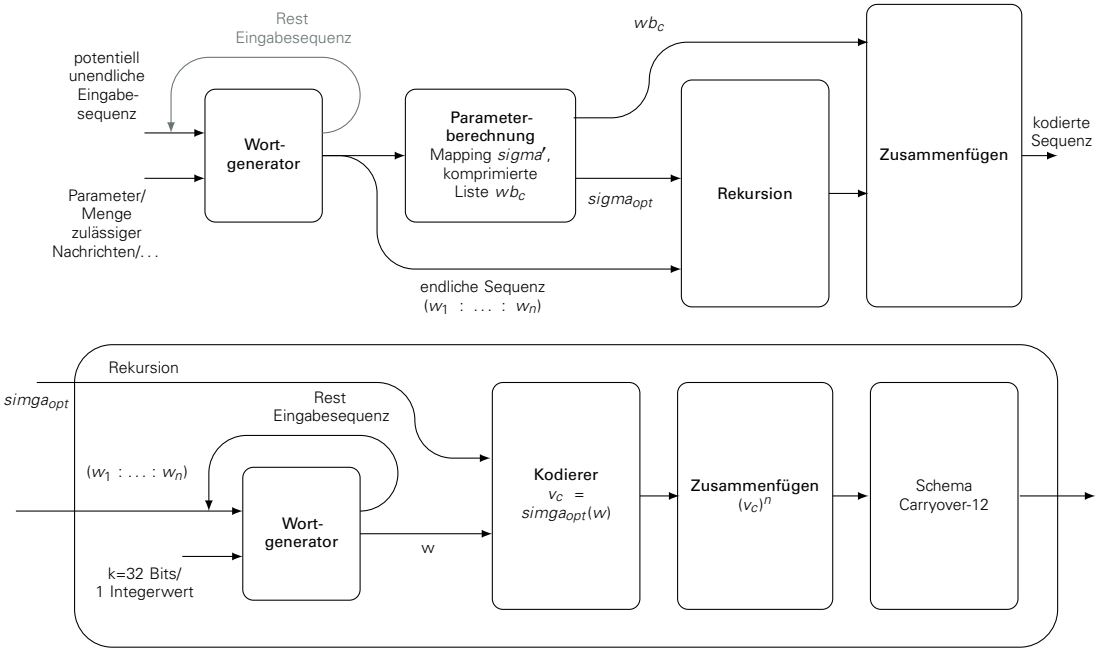
Das Kompressionsschema für den gesamten Algorithmus, die Kodierung der eigentlichen Werte, ist dann ein ganz normales Wörterbuchverfahren. Dieses zeigt Abbildung 4.25. Der Wortgenerator gibt eine endliche Sequenz aus, die Parameterberechnung die Wörterbuchabbildung sowie das komprimierte Wörterbuch. Das Modul des Zusammenfügens konkateniert die Größe des Wörterbuchs, das komprimierte Wörterbuch und die komprimierten Werte. Hier könnte man Carryover-12 und Wörterbuchabbildung verschachteln um eine tatsächlich schemakonforme Darstellung ohne mehrmalige Aufrufe zu erzwingen, dann müsste der erste Wortgenerator im Schema von Carryover-12 aber immer vorweggreifen, um die Anzahl der gemeinsam zu kodierenden Werte zu bestimmen.

4.8 ZUSAMMENFASSUNG

Das allgemeine Kompressionsschema ist durchaus geeignet, um eine Vielzahl verschiedener Algorithmen gut zu modularisieren und systematisch darzustellen. Durch den Austausch einzelner Module oder auch nur eingehender Parameter lassen sich häufig recht einfach verschiedene Algorithmen mit dem gleichen Kompressionsschema darstellen. Ein Beispiel hierfür ist, dass sich Varint-SU und Varint-PU nur im Zusammenfügen der Daten und Deskriptoren unterscheiden. Verschiedene zu erfüllende Teilaufgaben können voneinander abgekapselt gelöst und oft einzeln ausgetauscht werden. Einige Module und Modulgruppen tauchen in verschiedenen Algorithmen immer wieder auf, wie zum Beispiel die gesamte Rekursion, die das Binary Packing ausmacht, die sich in allen PFOR- und Simple-Algorithmen findet. Die Komplexität eines Algorithmus wird durch die Unterteilung in verschiedene, möglichst unabhängige kleinere Module, welche überschaubare Operationen ausführen, herabgesetzt. Die Strukturierung durch das Schema bildet eine gute Basis zur abstrakteren und

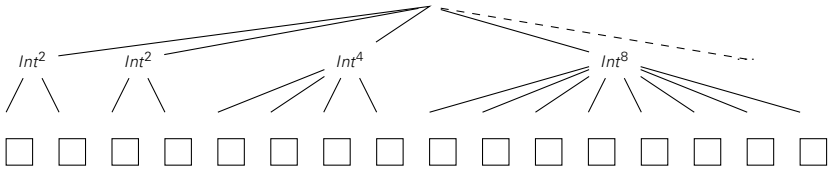


(a) Kompressionsschema Parameterberechnung

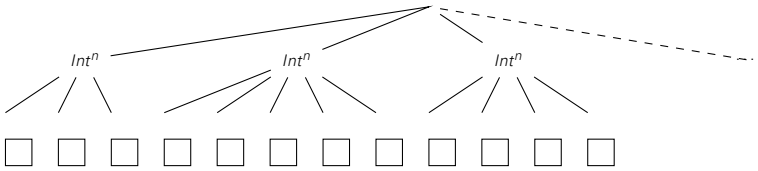


(b) Kompressionsschema für die Werte der sigmakodierten invertierten Dateien

Abbildung 4.25: Kompressionsschema sigmakodierte invertierte Dateien



(a) Hierarchische Datenorganisation Differenzkodierung



(b) Mögliche hierarchische Datenorganisation Carryover-12

Abbildung 4.26: Datenhierarchie bei der Differenzkodierung des Wörterbuches bei sigmakodierten invertierten Dateien

weniger implementierungsabhängigen Betrachtung von Kompressionsalgorithmen. Zu bedenken ist die verschieden starke Abhängigkeit zwischen Modulen, je nachdem, um welche Algorithmengruppe es sich handelt, zum Beispiel zwischen Wortgenerator und Parameterberechnung bei den Simple-Algorithmen. Es fällt auf, dass das Schema tatsächlich manchmal insgesamt mehrmals hintereinander aufgerufen werden muss und die Darstellung von sehr detailliert beschriebenen Algorithmen nicht trivial ist bzw. Details, wie im Nachhinein berechnete Werte, schwer abzubilden sind.

Interessant ist die Frage, welche Komprimierungseigenschaften anhand des Kompressionsschemas eines Algorithmus ablesbar sind, und andersherum, welche Eigenschaften bestimmte Modularisierungsmuster implizieren, was im nächsten Kapitel betrachtet werden soll.

5 EIGENSCHAFTEN VON KOMPRIMIERUNGSMETHODEN

Anhand verschiedenster Eigenschaften versucht man Aussagen über Kompressionsalgorithmen zu treffen und sie zu klassifizieren, um grobe Unterscheidungsmerkmale auszumachen. Dieses Kapitel widmet sich dem Versuch Zusammenhänge zwischen eher grundsätzlichen Merkmalen von Kompressionsalgorithmen und dem Kompressionsschema aufzuzeigen. Beispiele für solche Merkmale sind die Anpassbarkeit an die Daten (Kapitel 5.1), die Anzahl der benötigten Pässe (Kapitel 5.2) oder auch, welche Art von Information genutzt wird (Kapitel 5.3). In der Literatur finden sich auch diverse Versuche die Art der Redundanz zu klassifizieren, was in Kapitel 5.4 betrachtet werden soll.

5.1 ANPASSBARKEIT

Leweler und Hirschberg ([LH87]) unterscheiden Kompressionsmethoden hinsichtlich ihrer Anpassbarkeit in statische, semiadaptive (hybride) und adaptive (dynamische) Verfahren im Kontext der Nachrichtenübertragung, welche hier bedeutungserhaltend in den Kontext der Datenspeicherung übertragen werden können. Diese Unterscheidung ist bedeutend für die Modularisierung dieser Methoden.

Definition 5.1.1 (*Statische Kompression*)

> *Bei einer statischen Kompressionsmethode ist die Abbildung aller Nachrichten (Wörter/Einheiten) auf Codewörter bereits festgelegt, bevor das erste Wort kodiert wird, sodass ein gegebenes Wort jedes Mal, wenn es in der Sequenz auftaucht, durch das gleiche Codewort repräsentiert wird.*

Statische Verfahren bestehen nur aus drei grundlegenden Elementen, einem Wortgenerator, einem Kodierer und einem zum Wortgenerator komplementären Modul, welches die einzelnen Wörter wieder zusammensetzt (siehe Abb. 5.1).

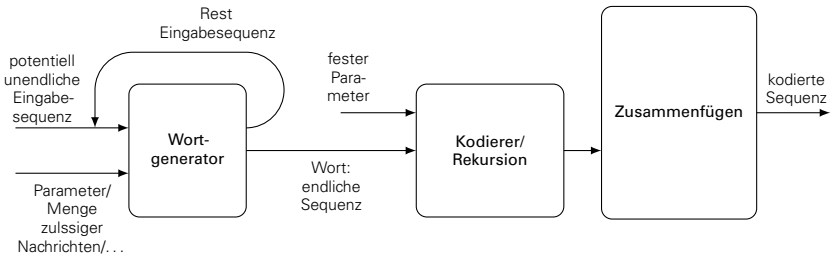


Abbildung 5.1: Statische Komprimierung

Definition 5.1.2 (*Semiadaptive Kompression*)

Semiadaptive Kompressionsmethoden passen ihr Datenmodell den Eingabedaten an. Die komprimierte Einheit muss das Datenmodell enthalten. Der Dekodierer dekomprimiert die Daten mithilfe des gespeicherten Datenmodells. Semiadaptive Methoden benötigen mehrere Pässe, wobei der erste zur Erstellung des Datenmodells genutzt wird.

Abbildung 5.2 zeigt ein Schema für semiadaptive Kompressionsmethoden. Daten können nur in mehreren Pässen verarbeitet werden, wenn die Eingabedatenmenge in endliche Teilsequenzen unterteilt werden kann. Der Wortgenerator erzeugt zunächst endliche Sequenzen von Daten, die am Ende wieder zu einem Datenstrom zusammengefügt werden. Die Erstellung des Datenmodells kann dabei die Berechnung eines Parameters über alle Werte oder die Erstellung eines Wörterbuches für eine endliche Sequenz sein. Das Schema für semiadaptive Kompressionsmethoden unterscheidet sich weiterhin darin von der statischen Kompression, dass anstelle des Kodierers das Schema rekursiv aufgerufen wird.

Definition 5.1.3 (*Adaptive Kompression*)

Ein Code ist adaptiv, wenn sich die Abbildung aus der Menge der Nachrichten in die Menge der Codewörter in Abhängigkeit von den bereits komprimierten Daten mit der Zeit verändert. Die komprimierten Daten müssen das aktuelle Datenmodell nicht enthalten. Der Kodierer komprimiert das nächste Wort und passt das Datenmodell entsprechend an. Der Dekodierer ist in der Lage, das aktuelle Datenmodell zur Dekompression einer Einheit zu generieren.

Adaptive Kompressionsalgorithmen unterscheiden sich darin, an welcher Stelle sie sich an die Daten anpassen. Beispielsweise ist Relative-10 ([AM05]) ein Algorithmus, welcher eine variable Anzahl von Zahlen mit einer gemeinsamen Bitweite in einem Blockcode mit 32 Bits speichert (zwei Deskriptorbits und 30 Datenbits). Dieser Algorithmus nutzt dabei die Eigenschaft seiner Eingabedaten, dass sich die Größe der Zahlen in einer Sequenz selten abrupt ändert. Es besteht dabei immer die Auswahl, für die nächste Sequenz die Bitweite um einen bestimmten Wert zu verringern um eine entsprechend größere Menge an Zahlen im Block kodieren zu können, die Bitweite beizubehalten, sie um einen bestimmten Wert zu erhöhen oder die nächste

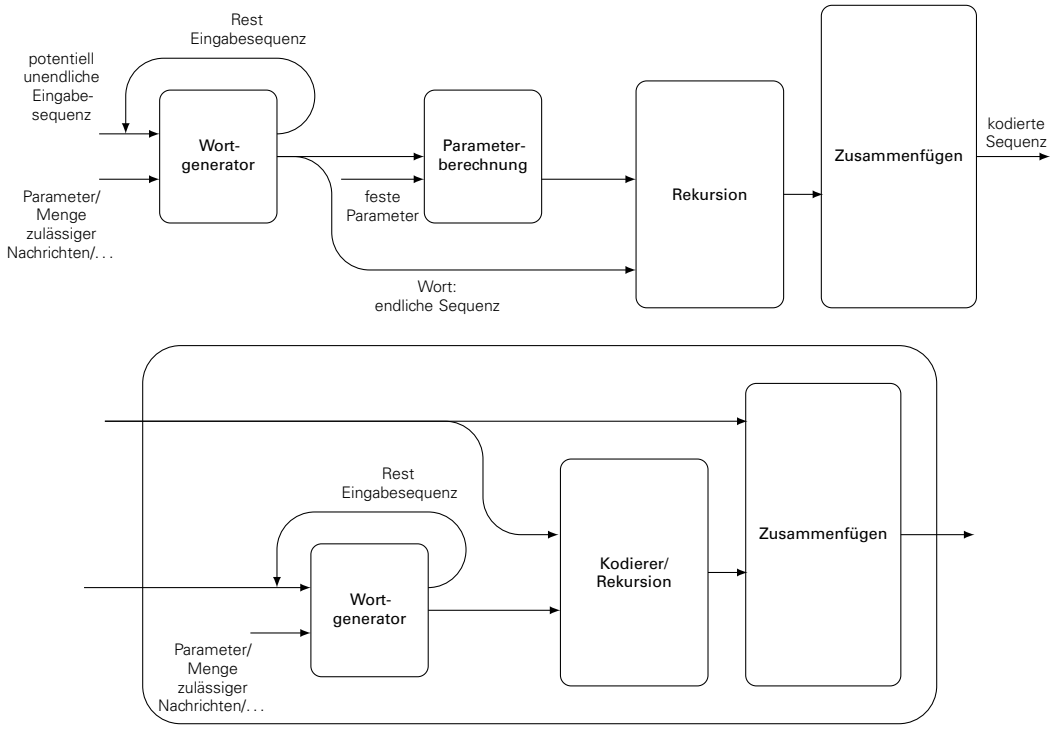


Abbildung 5.2: Semiadaptive Komprimierung

Zahl allein mit 30 Bit zu kodieren. Die Auswahl der Bitweiten und damit der Anzahl der durch den Wortgenerator zusammenzufassenden Werte hängt bei diesem Algorithmus von vorherigen Werten ab. Die Adaptivität betrifft in diesem Fall den Wortgenerator (siehe Abb. 5.3b). Zumeist jedoch beschränkt sich die Adaptivität eines Kompressionsalgorithmus auf die Parameterberechnung (siehe Abb. 5.3a). In beiden Arten adaptiv sind zum Beispiel die Lempel-Ziv-Algorithmen. Bei ihnen gibt es eine hohe Modulkopplung zwischen Wortgenerator und der Berechnung des erweiterten Wörterbuches als Parameterberechnung. Adaptive Algorithmen haben gemein, dass Daten zum Dekodieren sequentiell gelesen werden müssen.

Häufig findet sich in der Literatur der Begriff der lokal adaptiven Kompression, dessen Definition an dieser Stelle [SM10] entstammt.

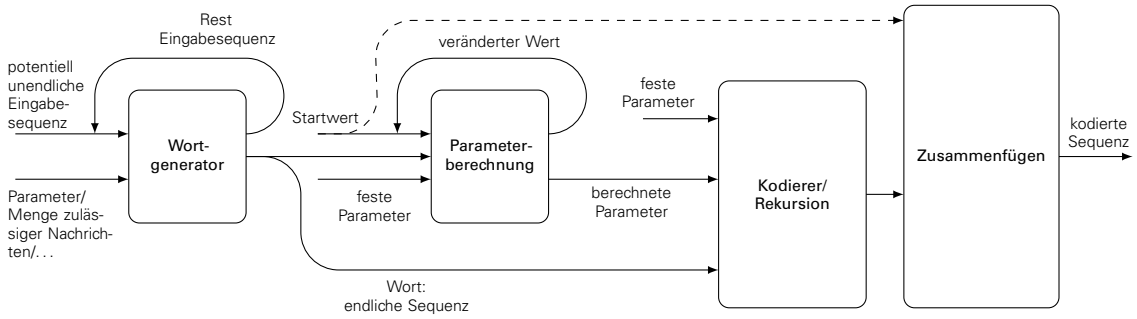
Definition 5.1.4 (*Lokal adaptive Kompression*)

> *Eine Kompressionsmethode ist lokal adaptiv, wenn sie sich lokal an den Eingabestrom anpasst und diese Adaption von Datenbereich zu Datenbereich ändert.*

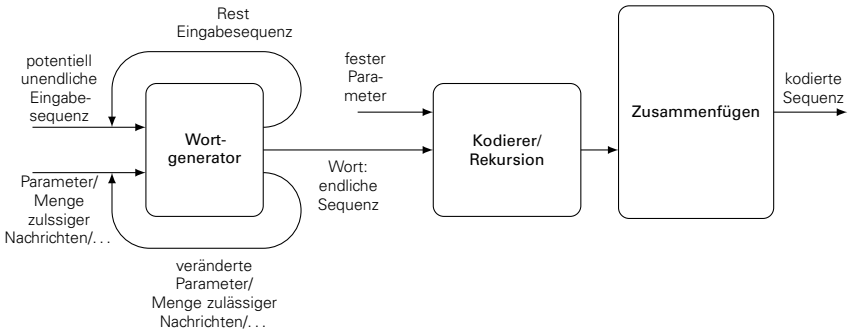
Das Kompressionsschema für solche Methoden benötigt wie semiadaptive Methoden eine Rekursion. Innerhalb der Rekursion existieren adaptive Module.

5.2 ANZAHL DER PÄSSE

Etwas schwieriger als die Anpassbarkeit ist prinzipiell der Zusammenhang zwischen Komprimierungsschema und Anzahl der Pässe darzulegen. Schon das Verständnis



(a) Adaptive Parameterberechnung



(b) Adaptiver Wortgenerator

Abbildung 5.3: Adaptive Komprimierung

des Begriffs *Pass* in verschiedenen Kontexten kann doch unterschiedlich sein. Im Kontext von Datenbanken und Anfrageausführung verbindet man normalerweise mit diesem Begriff die Frage, wie über die Daten iteriert werden muss. In Abgrenzung dazu wird der Begriff im Kontext von Streamingalgorithmen deutlich abstrakter verwendet. Ein Einpassalgorithmus erhält eine Eingabedatensequenz $w_1, w_2, w_3 \ldots$ und liest diese nur einmal von links nach rechts. Für die Berechnung eines komprimierten Wertes darf es keine Vorausgriffe auf Werte mit höherem Index geben. Ist dies der Fall, so wird der Algorithmus mit mehr als nur einen Pass ausgeführt. Mit dieser Definition gelingt im Grunde genommen nur eine Unterscheidung in Ein- und Mehrpassverfahren. Im Normalfall ist eine detailliertere Definition der Mehrpassverfahren nicht wichtig, man kennt Dreipassverfahren am ehesten aus der Videokompression. Im ersten Pass werden dabei Statistiken berechnet, auf deren Grundlage im zweiten Pass eine Videodatei komprimiert wird. Nutzung von Statistiken sind ein Vorgriff. Im zweiten Pass können, wenn gewünscht, aus den Quelldaten und den im ersten Pass berechneten Statistiken noch genauere Analysedaten berechnet werden. Da diese von den in ersten Passes berechneten Statistiken abhängig sind, handelt es sich wieder um einen Vorgriff. In diesem Sinne könnte man auch im Kontext dieser Arbeit eine genauere Passanzahl bestimmen, indem man Abhängigkeiten zwischen den statistisch berechneten und für die Kodierung relevanten Deskriptoren auf verschiedenen Hierarchieebenen, die bei semiadaptiven Verfahren in der Parameterberechnung

bestimmt werden, findet oder ausschließt. Ein Beispiel für ein Dreipassverfahren ist NewPFOR, bei dem die Bitweite pro Block festglegt wird. Von diesem Wert hängt ab, welche Werte als Ausnahmen behandelt werden und welche Bits eines Integerwertes als höhere Bits $b_\uparrow$. Höhere Bits der Ausnahmen werden mit Simple-16 kodiert, einem Zweipassverfahren. Wegen der Abhängigkeit des Kodierungsmodus als Deskriptor von der Bitweite als Deskriptor pro Block ist der ganze Algorithmus nach der obigen Betrachtungsweise ein Dreipassverfahren.

Zwei nacheinander ausgeführte Kompressionsschemen führen nicht zur Addition zur Passanzahl. Zwei nacheinander ausgeführte Einpassverfahren benötigen immer noch nur einen Pass. Ein Ein- und ein Zweipassverfahren, hintereinander ausgeführt, benötigen immer noch nur zwei Pässe. Das Schema an sich ist nur ein Modell für Kompressionsverfahren, welches hinreichend gut bestimmte Eigenschaften von Algorithmen und Algorithmenklassen abbildet. Es basiert auf strukturellen Überlegungen. Es kann mehrere mehr oder weniger plausible Möglichkeiten geben, einen Algorithmus zu modularisieren. Wenn man vom Kompressionsschema bzw. der hierarchischen Datenorganisation ganz klar auf die Anzahl der Pässe schließen könnte, müssten alle möglichen Kompressionsschemen, die für einen Algorithmus denkbar sind, zum gleichen Ergebnis führen, egal, ob man Mehrfachausführungen des Schemas möglicherweise einbauen oder weglassen kann oder, falls möglich, Rekursionstiefen verändert. Allerdings lohnt es sich, den Zusammenhang zwischen Anpassbarkeit und Anzahl der Pässe sowie der Datenorganisation und der benötigten Passanzahl zu betrachten.

Schluss von der Passanzahl über die Anpassbarkeit auf das Kompressionsschema Bei Einpassmethoden schließen sich semiadaptive Verfahren prinzipiell aus, das heißt, dass generell nur statische oder adaptive Methoden in Frage kommen. Mehrpassmethoden zeichnen sich durch eine Rekursion aus.

Schluss vom Kompressionsschema auf die Passanzahl Kompressionsschemata ohne Rekursionen können nur Einpassverfahren abbilden. Die Anzahl der Rekursionen muss allerdings nicht mit der Passanzahl einhergehen, sie korreliert aber mit der hierarchischen Datenorganisation eines Algorithmus. Jeder Knoten, jede endliche Sequenz kann sich durch einen berechneten Parameter, einem Deskriptor d auszeichnen. Jeder Kindknoten eines Knotens kann wiederum für die Berechnung eines Parameters d' für eine Teilsequenz stehen. Sind d und d' voneinander unabhängig, macht es strukturell betrachtet Sinn, eine weitere Rekursion einzuführen, praktisch können diese Parameter aber in einem statt zwei Pässen berechnet werden, weswegen die Anzahl der Rekursionen nicht mit der Anzahl der Pässe korrelieren muss.

5.3 GENUTZTE INFORMATION

Verschiedene Algorithmen und Algorithmenklassen nutzen verschiedene Arten von Information um eine möglichst gute Kompression zu ermöglichen. Jede Entropiekodierung basiert auf der Annahme, dass Daten nicht gleichverteilt sind. Diese Verteilungen können statistisch berechnet werden oder als Kontextinformation vorliegen. Viele Kodierungen basieren auf der Tatsache, dass der gegebene, möglicherweise unendliche Wertebereich einschränkbar ist und welche Werte überhaupt vorkommen und welche nicht, was als Information wiederum aus berechneten Statistiken oder aus dem Kontext stammt.

Statistik Statistiken können im Modul der Parameterberechnung erstellt werden. Beispiele sind die Berechnung eines Wörterbuches für variabel lange Codes bei der Huffmankodierung oder, um den Wertebereich zu beschränken, Referenzwert und Bitweite beim FOR oder die Berechnung des Wörterbuches bei Bitvektoren. Statistische Information bedeutet, dass aus einer endlichen Sequenz ein oder mehrere Werte berechnet werden, die in einen Kodierer, einen Wortgenerator, ein Splitmodul oder in eine weitere Parameterberechnung eingehen.

Kontext Was man als Kontextinformation bezeichnen möchte, ist streitbar. Kontextinformationen können zum einen im Kompressionsschema durch in ein Modul eingehende Informationen nicht definierten Ursprungs sein, die beim Dekodieren bekannt sind und genutzt werden, ohne dass sie mit gespeichert werden. Man kann darunter auch Wissen über die Daten verstehen, zum Beispiel über die Art der Redundanz, zum Beispiel auch Informationen über die zu erwartende Datenverteilung oder Wissen über Speichergrößen. Kontextinformation kann nicht nur Parameter für einzelne Module, sondern die gesamte Wahl des Kompressionsalgorithmus beeinflussen.

5.4 ART DER DATEN UND ARTEN VON REDUNDANZ

Gewisse Eigenschaften der Daten sind für die Nutzung verschiedener Muster notwendig. So setzen FOR und DELTA metrische Eingabedaten voraus. Ordnungserhaltende Wörterbuchverfahren benötigen plausiblerweise Daten, auf denen eine Ordnung bzw. Rangfolge definiert ist. Für allgemeine Wörterbuchverfahren, Bitvektorverfahren, RLE und Symbolunterdrückung müssen die Eingabedaten weder metrisch sein noch einer Ordnung unterliegen. Nur eine Äquivalenzrelation ist notwendig. Damit ergeben sich drei Klassen von vorausgesetzten Dateneigenschaften. Die verschiedenen vorausgesetzten Eigenschaften machen den Versuch Redundanz zu klassifizieren nicht einfach.

[Wel84] versucht sich daran vier Kategorien von Redundanz für Zeichenketten (ohne Ordnungsrelation auf der Menge der Zeichen) zu erstellen, die im Folgenden kurz erläutert werden.

Ungleiche Zeichenverteilung Es ist bekannt, dass verschiedene Werte eines Wertevorrates unabhängig von Vorgängerwerten und Position in der Sequenz verschiedene Auftrittswahrscheinlichkeiten besitzen. Die Zeichenverteilung ist aus dem Kontext bekannt oder wurde statistisch berechnet. In letzterem Fall betrifft die Redundanz nur eine endliche Teilsequenz.

Zeichenwiederholungen Bei dieser Redundanzkategorie ist das Wissen vorhanden, dass die Wahrscheinlichkeit, dass einem Zeichen noch einmal das gleiche Zeichen folgt, größer ist als die, dass ein davon verschiedenes Zeichen an der nächsten Position auftaucht. Dies kann für ein bestimmtes Zeichen oder aber für alle Zeichen eines Wertevorrates gelten.

Ungleiche Häufigkeiten von Mustern Manche endlichen Sequenzen tauchen mit erhöhter Wahrscheinlichkeit auf. Um diese Redundanz zu vermeiden, kann man solche endlichen Sequenzen mit weniger Bits kodieren.

Positionale Redundanz Mit positionaler Redundanz ist das vorhersagbare Auftauchen von Zeichen an bestimmten Stellen innerhalb einer Sequenz gemeint.

Die letzte Kategorie setzt eine Menge von endlichen Sequenzen voraus, deren Werte an bestimmten Positionen sich vergleichen lassen, so dass nur Algorithmen mit Rekursionen im Kompressionsschema diese Art von Redundanz reduzieren können. Die anderen drei Kategorien setzen nur einen Datenstrom voraus, über den gewisse Kontextinformationen vorliegen. Für all diese Klassifikationen müssen die Daten lediglich auf Gleichheit untersuchbar sein. In [Wel84] bemerkt der Autor jedoch selbst, dass diese Klassifikation viele Überlappungen zulässt. Bei der Definition dieser Kategorien wurden spezifische Eigenschaften metrischer Werte sowie semiadaptiver Verfahren völlig außer Acht gelassen. Um semiadaptiven Verfahren gerecht zu werden, könnte man die Definition positionaler Redundanz weiterfassen. Diese Kategorie kann so definiert werden, dass Wahrscheinlichkeiten für das Auftreten einzelner Zeichen in einer Sequenz positionsabhängig sind.

Es gibt immer einen Zusammenhang zwischen dem Vorhandensein von Redundanz und angenommenen ungleichen Wahrscheinlichkeitsverteilungen aufgrund von Kontextwissen oder berechneten und ungleichen Häufigkeiten. Solche statistische Information wirkt im Kompressionsschema innerhalb von Rekursionen auch wie eine Wahrscheinlichkeitsverteilung aus Kontextwissen heraus. Es lassen sich folgende

drei grundsätzliche Kategorien als Grundlage für angenommene Wahrscheinlichkeiten oder berechnete Häufigkeiten ausmachen: generell eine ungleiche Zeichenverteilung, Abhängigkeiten der Wahrscheinlichkeiten für das Auftreten von Werten von vorangegangenen Werten und Abhängigkeiten der Wahrscheinlichkeiten für das Auftreten von Werten von der Position innerhalb einer Sequenz.

Ungleiche Zeichenverteilung Diese Kategorie ist die gleiche wie die oben beschriebene.

Wahrscheinlichkeitsabhängigkeiten von Werten In dieser Kategorie werden alle Redundanzformen zusammengefasst, bei welchen die Auftrittswahrscheinlichkeit eines Wertes von allen oder einer endlichen Menge an Vorgängerwerten abhängt. Beispielsweise könnte bekannt sein, dass in einer Sequenz häufiger gleiche Werte direkt nacheinander auftreten.

Wahrscheinlichkeitsabhängigkeit von Positionen Manchmal ist das Wissen vorhanden, dass an verschiedenen Positionen innerhalb einer Sequenz ungleiche Auftrittswahrscheinlichkeiten vorliegen. Beispielsweise könnte jede Sequenz mit dem gleichen Wert starten.

Unabhängig von dieser Einteilung geht es also immer um ungleiche Auftrittswahrscheinlichkeiten von Werten. Man kann weiterhin noch unterscheiden ob dieses Wissen so aussieht, das die Wahrscheinlichkeiten für jeden Wert einzeln definiert sind oder ob nur zwei Kategorien aufgestellt werden: Die Kategorie der Werte, die nicht auftreten können und die Kategorie der Werte, die auftreten können und über die meist kein Wissen über Unterschiede hinsichtlich der Auftrittswahrscheinlichkeiten vorliegt. Beispielsweise ist bei Bitvektoren bekannt, welche Werte vorkommen dürfen und welche nicht. Da keine Aussagen über Auftrittswahrscheinlichkeiten genutzt werden sollen bzw. diese Art von Redundanz gewollt nicht reduziert werden soll, können alle Werte mit gleich langen Codewörtern kodiert werden.

In Abbildung 5.4 ist ein Ordnungsversuch skizziert. Verschiedene in dieser Arbeit beschriebene Muster zur Redundanzreduktion sind innerhalb des Ordnungssystems positioniert. Die drei festgelegten Kategorien der ungleichen Zeichenverteilung, der Wahrscheinlichkeitsabhängigkeiten von vorangegangenen Werten und in der Wahrscheinlichkeitsabhängigkeiten von Positionen innerhalb einer Sequenz sind durch die Sektoren dargestellt. Im inneren Kreis befinden sich Redundanzarten und ihnen zugeordnete Techniken, bei denen es generell um ungleiche Verteilungen geht. Der äußere Kreis enthält die Begriffe für Redundanzarten, bei denen das Wissen um Werte, die nicht auftreten können, im Vordergrund steht.

Sind Zeichen unabhängig von der Position in der Sequenz und von vorangehenden Werten unterschiedlich verteilt, bieten sich als Wörterbuchverfahren Entropiekodie-

rungen an. Teilweise auch das Symbolunterdrückungsmuster, wenn ein bestimmtes Zeichen sehr viel häufiger zu erwarten ist als andere. Techniken wie FOR, bei dem Werte, die kleiner als der Referenzwert sind, nicht erlaubt sind, Bitvektoren, die viele Werte per se ausschließen oder die Festlegung der Bitweite beim Binary Packing, die eine obere Grenze für den Wertebereich definiert, legen nur zwei Kategorien von Werten fest: solche, die vorkommen und kodiert werden können und solche, die nicht vorkommen. Bei Sequenzen von natürlichen Zahlen geht man davon aus, dass kleinere Werte häufiger vorkommen als größere. Ist eine natürliche Zahlen binär als endliche Sequenz von Bits kodiert, so ist die Wahrscheinlichkeit, dass an den vorderen Positionen eine 0 auftaucht, deutlich höher als das Auftauchen einer 1. Nullenunterdrückung setzt also auch an der Reduktion positionsabhängiger Redundanz an. Beim Binary Packing ist aus dem Kontext bekannt, dass eine bestimmte Anzahl von führenden Nullen bei allen Werten einer endlichen Sequenz vorhanden ist. Das Auftauchen von Nullen an diesen Stellen ist sicher. Für diese Stellen gibt es nur zwei Kategorien von Werten, Nullen, die definitiv dort auftreten und Einsen, die dort definitiv nicht auftreten. Diese Sicherheit des Auftauchens ist in [Wel84] mit positionaler Redundanz gemeint.

Orange dargestellt ist die Kategorie der werteabhängigen Wahrscheinlichkeiten. Hierein fallen die Kategorien der ungleichen Häufigkeiten von Mustern und der Zeichenwiederholung aus [Wel84]. Einfaches RLE reduziert Redundanz bei erhöhter Wahrscheinlichkeit von Zeichenwiederholungen, Nullenunterrückung und Symbolunterdrückung mit RLE bei erhöhter Wahrscheinlichkeit von Zeichenwiederholungen bei bestimmten Zeichen. Ungleiche Wahrscheinlichkeiten für das Auftreten von Sequenzen schließen bei sortiert vorliegenden Daten das Auftreten bestimmter Sequenzen aus. Bei einer Liste von sortiert vorliegenden Zahlen kann das Bigramm (2,1) nicht auftauchen. Hier findet die Differenzkodierung einen Ansatzpunkt zur Redundanzreduktion.

5.5 ZUSAMMENFASSUNG

Eine erschöpfende Darlegung von Eigenschaften von Komprimierungsmethoden ist kaum möglich. Im Rahmen dieser Arbeit kann nur im Ansatz auf verschiedene Eigenschaften eingegangen werden. Gerade aus der geforderten Art der Anpassbarkeit einer Kompressionsmethode lässt sich ein passendes Grundgerüst bestimmen. Zusammenhänge zwischen Passzahlen (und damit verbunden Fragen nach einer möglichst geringen Kompressionszeit) und Eigenschaften des modularisierten Kompressionsschemas sind noch nicht befriedigend stark herstellbar. Auch der Abschnitt zur genutzten Information umreißt nur sehr grob die Unterscheidung in statistische Berechnungen aufgrund der vorliegenden Daten und Kontextinformation. Die Unterteilung in verschiedene Arten der Redundanz der Daten ist strukturell und inhaltlich valide, muss jedoch weiter durchdacht und auf praktischen Nutzen geprüft werden.

Abbildung 5.4: Ordnungsversuch für verschiedene Arten von Redundanz

6 ZUSAMMENFASSUNG UND AUSBLICK

Das in dieser Arbeit neu erstellte modularisierende Kompressionsschema ist ausreichend allgemein, um viele verschiedene Kompressionsalgorithmen darzustellen. Gängige Kompressionsalgorithmen sind teilweise so originell, dass das einfache Kompressionsschema zur Darstellung Erweiterungen benötigt, wie beispielsweise bei einer Umordnung von Daten. Durch verschiedene Parametereingaben können Ausgaben und ganze Verfahren variiert werden. Durch die Möglichkeit Module sehr passend zusammenzustellen und mit Inhalt zu füllen, ergibt sich ein mächtiges Werkzeug für den automatisierten Bau von Algorithmen. Das Kompressionsschema bietet eine Grundlage und eröffnet die Möglichkeit, für einen gegebenen Kontext sehr gezielt speziell zugeschnittene Algorithmen mit bestimmten Eigenschaften wie zum Beispiel der Art der Anpassbarkeit zusammenzubauen. Weiterhin können verschiedene Muster wie FOR, DELTA, Bitvektoren, Symbolunterdrückung oder Lauflängenkodierung an den Kontext angepasst eingesetzt werden und das auf verschiedensten Ebenen miteinander kombiniert. Möglicherweise ergeben sich Beschränkungen dadurch, dass sich spezielle Aspekte nicht durch das Kompressionsschema modellieren lassen. Aufgefallen sind bisher Parameterberechnungen, die nach der Kodierung stattfinden wie Größen- und Positionsangaben.

Für die Fortführung dieses Gedankens ist es notwendig, einen wesentlich stärkeren Zusammenhang zwischen Kontextwissen und passender Schemazusammenstellung sowie passenden Parametereingaben herzustellen. Auch eine praktische Umsetzung sollte den Nutzen eines individuell an den Kontext angepassten Kompressionsalgorithmus zeigen und überprüfen, inwieweit sich Vorteile im Vergleich zu einer einfachen Auswahl aus einer verhältnismäßig kleinen Menge an Kompressionsalgorithmen ergeben, um den Flaschenhals des Hauptspeicherzugriffs zu verringern. Der automatisierte Zusammenbau eines Kompressionsalgorithmus wird bei unklaren Entscheidungsfällen möglicherweise zum Optimierungsproblem, dessen Berechnung umso mehr Zeit kostet, je größer der Lösungsraum gewählt wird. Der Frage,

inwieweit sich der Aufwand lohnt und durch eine bessere Anpassung Nutzen bringt, muss in Zukunft nachgegangen werden.

ABBILDUNGSVERZEICHNIS

LITERATURVERZEICHNIS

[AGOS13] ARROYUELO, Diego ; GONZÁLEZ, Senén ; OYARZÚN, Mauricio ; SEPULVEDA, Victor: Document Identifier Reassignment and Run-length-compressed Inverted Indexes for Improved Search Performance. In: *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA : ACM, 2013 (SIGIR '13). – ISBN 978–1–4503–2034–4, 173–182

[ALM96] ANTOSHENKOV, Gennady ; LOMET, David B. ; MURRAY, James: Order Preserving Compression. In: SU, Stanley Y. W. (Hrsg.): *ICDE*, IEEE Computer Society, 1996. – ISBN 0–8186–7240–4, 655-663

[AM05] ANH, Vo N. ; MOFFAT, Alistair: Inverted Index Compression Using Word-Aligned Binary Codes. In: *Inf. Retr.* 8 (2005), Januar, Nr. 1, 151–166. http://dx.doi.org/10.1023/B:INRT.0000048490.99518.5c. – DOI 10.1023/B:INRT.0000048490.99518.5c. – ISSN 1386–4564

[AM06] ANH, Vo N. ; MOFFAT, Alistair: Improved Word-Aligned Binary Compression for Text Indexing. In: *IEEE Trans. on Knowl. and Data Eng.* 18 (2006), Juni, Nr. 6, 857–861. http://dx.doi.org/10.1109/TKDE.2006.99. – DOI 10.1109/TKDE.2006.99. – ISSN 1041–4347

[AM10] ANH, Vo N. ; MOFFAT, Alistair: Index Compression Using 64-bit Words. In: *Softw. Pract. Exper.* 40 (2010), Februar, Nr. 2, 131–147. http://dx.doi.org/10.1002/spe.v40:2. – DOI 10.1002/spe.v40:2. – ISSN 0038–0644

[Aro77] ARONSON, Jules: Computer science and technology: data compression — a comparison of methods / Department of Commerce, National Bureau of Standards, Institute for Computer Sciences and Technology. Version: Juni 1977. http://ucblibraries.colorado.edu/circulation/forms/ericrequest.htm;http://www.eric.ed.gov/contentdelivery/servlet/ERICServlet?accno=ED149732. Washington, DC, USA, Juni

1977 (500-12). – NBS special publication. – iv + 31 + 1 S. – ISSN 0083–1883. – ERIC Document Number: ED149732

[asf04] *Apache Software Foundation. Lucene 1.4.3 documentation*. http://lucene.apache.org/java/1_4_3/fileformats.html. Version: 2004

[AZW+11] AO, Naiyong ; ZHANG, Fan ; WU, Di ; STONES, Douglas S. ; WANG, Gang ; LIU, Xiaoguang ; LIU, Jing ; LIN, Sheng: Efficient Parallel Lists Intersection and Index Compression Algorithms Using Graphics Processing Units. In: *Proc. VLDB Endow.* 4 (2011), Mai, Nr. 8, 470–481. http://dx.doi.org/10.14778/2002974.2002975. – DOI 10.14778/2002974.2002975. – ISSN 2150–8097

[BCC10] BÜTTCHER, Stefan ; CLARKE, Charles ; CORMACK, Gordon V.: *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press, 2010. – ISBN 0262026511, 9780262026512

[CMS09] CROFT, Bruce ; METZLER, Donald ; STROHMAN, Trevor: *Search Engines: Information Retrieval in Practice*. 1st. USA : Addison-Wesley Publishing Company, 2009. – ISBN 0136072240, 9780136072249

[CP90] CUTTING, D. ; PEDERSEN, J.: Optimization for Dynamic Inverted Index Maintenance. In: *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA : ACM, 1990 (SIGIR '90). – ISBN 0–89791–408–2, 405–411

[DCS+10] DELBRU, Renaud ; CAMPINAS, Stephane ; SAMP, Krystian ; TUMMARELLO, Giovanni ; DANGAN, Lower ; DELBRU, Renaud ; CAMPINAS, Stephane ; SAMP, Krystian ; TUMMARELLO, Giovanni: *ADAPTIVE FRAME OF REFERENCE FOR COMPRESSING INVERTED LISTS*. 2010

[DCT12] DELBRU, Renaud ; CAMPINAS, Stephane ; TUMMARELLO, Giovanni: Searching Web Data: An Entity Retrieval and High-performance Indexing Model. In: *Web Semant.* 10 (2012), Januar, 33–58. http://dx.doi.org/10.1016/j.websem.2011.04.004. – DOI 10.1016/j.websem.2011.04.004. – ISSN 1570–8268

[Dea09] DEAN, Jeffrey: Challenges in building large-scale information retrieval systems: invited talk. In: BAEZA-YATES, Ricardo A. (Hrsg.) ; BOLDI, Paolo (Hrsg.) ; RIBEIRO-NETO, Berthier A. (Hrsg.) ; CAMBAZOGLU, Berkant B. (Hrsg.): *WSDM*, ACM, 2009. – ISBN 978–1–60558–390–7, 1

[DRCZ07] DEVEAUX, Jean-Paul ; RAU-CHAPLIN, Andrew ; ZEH, Norbert: Adaptive Tuple Differential Coding. In: *Proceedings of the 18th International Conference on Database and Expert Systems Applications*. Berlin,

Heidelberg : Springer-Verlag, 2007 (DEXA'07). – ISBN 3–540–74467–3, 978–3–540–74467–2, 109–119

[Gro95] GROSSMAN, David A.: *Integrating Structured Data and Text: A Relational Approach*. Fairfax, VA, USA, Diss., 1995. – UMI Order No. GAX96-19482

[Hea72] HEAPS, H. S.: Storage analysis of a compression coding for a document database. (1972), February, S. 47–61

[Hé05] HÉMAN, Sándor: *Super-Scalar Database Compression between RAM and CPU Cache*, Universiteit van Amsterdam, Masterarbeit, 2005

[LB12] LEMIRE, Daniel ; BOYTSOV, Leonid: Decoding billions of integers per second through vectorization. In: *CoRR* abs/1209.2137 (2012)

[LH87] LELEWER, Debra A. ; HIRSCHBERG, Daniel S.: Data Compression. In: *ACM Comput. Surv.* 19 (1987), September, Nr. 3, 261–296. http://dx.doi.org/10.1145/45072.45074. – DOI 10.1145/45072.45074. – ISSN 0360–0300

[mid01] *MIDI Manufacturers Association. MIDI 1.0 Specification*. 1982–2001

[MRS08] MANNING, Christopher D. ; RAGHAVAN, Prabhakar ; SCHÜTZE, Hinrich: *Introduction to Information Retrieval*. New York, NY, USA : Cambridge University Press, 2008. – ISBN 0521865719, 9780521865715

[NR97] NG, Wee K. ; RAVISHANKAR, Chinya V.: Block-Oriented Compression Techniques for Large Statistical Databases. In: *IEEE Trans. on Knowl. and Data Eng.* 9 (1997), März, Nr. 2, 314–328. http://dx.doi.org/10.1109/69.591455. – DOI 10.1109/69.591455. – ISSN 1041–4347

[Reg81] REGHBATI, Hassan K.: An Overview of Data Compression Techniques. In: *IEEE Computer* 14 (1981), Nr. 4, 71-75. http://dblp.uni-trier.de/db/journals/computer/computer14.html#Reghbati81

[RH93] ROTH, Mark A. ; HORN, Scott J. V.: Database Compression. In: *SIGMOD Record* 22 (1993), Nr. 3, 31-39. http://dblp.uni-trier.de/db/journals/sigmod/sigmod22.html#RothH93

[SGL10] SCHLEGEL, Benjamin ; GEMULLA, Rainer ; LEHNER, Wolfgang: Fast Integer Compression Using SIMD Instructions. In: *Proceedings of the Sixth International Workshop on Data Management on New Hardware*. New York, NY, USA : ACM, 2010 (DaMoN '10). – ISBN 978–1–4503–0189–3, 34–40

[SGR+11] STEPANOV, Alexander A. ; GANGOLLI, Anil R. ; ROSE, Daniel E. ; ERNST, Ryan J. ; OBEROI, Paramjit S.: SIMD-based Decoding of Posting Lists.

In: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. New York, NY, USA : ACM, 2011 (CIKM '11). – ISBN 978–1–4503–0717–8, 317–326

[SM10] SALOMON, David ; MOTTA, Giovanni: *Handbook of Data Compression (5. ed.).* Springer, 2010. – ISBN 978–1–84882–902–2

[SV10] SILVESTRI, Fabrizio ; VENTURINI, Rossano: VSEncoding: Efficient Coding and Fast Decoding of Integer Lists via Dynamic Programming. In: *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*. New York, NY, USA : ACM, 2010 (CIKM '10). – ISBN 978–1–4503–0099–5, 1219–1228

[TS07] TROTMAN, Andrew ; SUBRAMANYA, Vikram: Sigma Encoded Inverted Files. In: *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*. New York, NY, USA : ACM, 2007 (CIKM '07). – ISBN 978–1–59593–803–9, 983–986

[Wel84] WELCH, T. A.: A Technique for High-Performance Data Compression. In: *Computer* 17 (1984), Juni, Nr. 6, 8–19. http://dx.doi.org/10.1109/MC.1984.1659158. – DOI 10.1109/MC.1984.1659158. – ISSN 0018–9162

[Wil91] WILLIAMS, Ross N.: *Adaptive Data Compression*. 1991

[YDS09] YAN, Hao ; DING, Shuai ; SUEL, Torsten: Inverted Index Compression and Query Processing with Optimized Document Ordering. In: *Proceedings of the 18th International Conference on World Wide Web*. New York, NY, USA : ACM, 2009 (WWW '09). – ISBN 978–1–60558–487–4, 401–410

[ZHNB06] ZUKOWSKI, Marcin ; HEMAN, Sandor ; NES, Niels ; BONCZ, Peter: Super-Scalar RAM-CPU Cache Compression. In: *Proceedings of the 22Nd International Conference on Data Engineering*. Washington, DC, USA : IEEE Computer Society, 2006 (ICDE '06). – ISBN 0–7695–2570–9, 59–

[ZlJ93] ZANDI, A. ; IYER, Balakrishna R. ; JR., Glen G. L.: Sort Order Preserving Data Compression for Extended Alphabets. In: STORER, James A. (Hrsg.) ; COHN, Martin (Hrsg.): *Data Compression Conference*, IEEE Computer Society, 1993. – ISBN 0–8186–3392–1, 330-339

[ZLS08] ZHANG, Jiangong ; LONG, Xiaohui ; SUEL, Torsten: Performance of Compressed Inverted List Caching in Search Engines. In: *Proceedings of the 17th International Conference on World Wide Web*. New York, NY, USA : ACM, 2008 (WWW '08). – ISBN 978–1–60558–085–2, 387–396