Technische Universität Kaiserslautern
Fachbereich Informatik
AG Datenbanken und Informationssysteme
Prof. Dr.-Ing. Dr. h. c. Theo Härder

# Speicherungsstrukturen für native XML-Datenbanksysteme

**Diplomarbeit**

von

Markus Wagner

Betreuer:

Dipl.-Inform. Michael P. Haustein

September 2005

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur angefertigt habe.

Kaiserslautern, den 24. Oktober 2005

---

Markus Wagner

# Inhaltsverzeichnis

# Kapitel 1 Einleitung

Die „Extensible Markup Language“ (XML) [20], ursprünglich für großangelegte elektronische Veröffentlichungen gedacht, spielt beim elektronischen Datenaustausch eine immer wichtigere Rolle. Dies führt dazu, dass eine immer größere Anzahl von XML-Dokumenten verwaltet werden muss. Daneben ist es gerade bei großen Dokumenten wünschenswert, dass mehrere Benutzer die Möglichkeit haben, gleichzeitig auf die Dokumente zugreifen und Änderungen vornehmen zu können. Diese Anforderungen führen unausweichlich zu einem XML-Datenbankmanagementsystem (XDBMS).

Viele der bisherigen Forschungsbemühungen versuchen, XML-Dokumente auf relationale Datenbanksysteme abzubilden [10, 11, 12]. Der Versuch, relationale Datenbanksysteme für diese Aufgabe zu verwenden, ist einleuchtend, da diese durch ihre lange Entwicklungszeit perfektioniert werden konnten. Zwar erbt man durch die Übernahme dieser Systeme die für den Transaktionsschutz notwendigen Eigenschaften Atomarität, Konsistenz, Isolation und Dauerhaftigkeit (ACID-Eigenschaften) [2]. Doch diese Methode hat auch entscheidende Nachteile. Wenn das komplette Dokument als CLOB in einem Datensatz abgelegt wird, kann keine differenzierte Transaktionsverwaltung stattfinden, da beim Zugriff immer nur das gesamte Dokument gesperrt werden kann. Zerlegt man dagegen das Dokument in seine Einzelteile und legt diese Einzelteile in verschiedenen Tabellen ab, erfordert das Einfügen eines einzelnen XML-Elements in das XML-Dokument mehrere Einfügungen in verschiedenen relationalen Tabellen.

Das Problem der Abbildung von XML-Daten auf ein relationales Modell ist in den sehr unterschiedlichen Strukturen begründet. Während relationale Daten innerhalb einer Tabelle flach strukturiert sind, sind XML-Dokumente hierarchisch strukturiert. Die hierarchische Struktur muss folglich „künstlich“ auf die flache Struktur abgebildet werden. Um dabei keine Informationen zu verlieren, werden umfangreiche Zusatzinformationen benötigt. Wenn in einem XML-Dokument ein Teilbaum gesperrt werden muss, ist das aufgrund der hierarchischen Struktur relativ leicht möglich. Um Phantome zu vermeiden, werden dagegen vom relationalen Datenbanksystem oft ganze Tabellen gesperrt, so dass auch die Teile des Dokuments gesperrt werden, die von der Transaktion gar nicht betroffen sind.

Aufgrund dieser Nachteile sollten XML-Daten in einem Datenbanksystem gespeichert werden, das die Verarbeitung von XML-Dokumenten auf einem dafür zugeschnittenen Synchronisationsmodell vornimmt. Ein solches Datenbanksystem ist der *XML Transaction Coordinator*, ein Prototyp eines nativen XML-Datenbanksystems, dessen Transaktionsmodell und Speicherungsstrukturen an die Anforderungen der XML-Datenverarbeitung angepasst sind.

Der Aufbau des *XTC* ist an das Fünf-Schichten-Architekturmodell angelehnt und wird in Kapitel 2 beschrieben. Als Grundlage zur Datenhaltung und Anfrageverarbeitung dient das taDOM-Transaktionsmodell, in dem der DOM-Baum zum taDOM-Baum erweitert wird und Operationen zur Datenverarbeitung definiert werden. Die Daten des XML-Dokuments werden im so genannten *Dokumentcontainer* abgelegt. Zusätzlich wird der Elementindex zur Verwaltung der Elementknoten angelegt.

Jeder Knoten des taDOM-Baums erhält eine eindeutige ID, die so genannte DeweyID, mit deren Hilfe Knoten indexiert werden können und auf deren Grundlage die hierarchischen Sperrprotokolle aufbauen, da sie die dafür nötige Eigenschaft besitzt, Vorfahren anhand der DeweyID des Kontextknotens zu bestimmen. Weitere Vorteile der DeweyIDs sowie deren Vergabe und Kodierung werden in Kapitel 3 diskutiert.

Im vierten Kapitel werden zum einen die Indexstrukturen vorgestellt, die im XTC-Server verwendet werden. Zum anderen wird beschrieben, wie die Parallelität beim Zugriff auf diese Indexstrukturen durch vorzeitige Freigabe von Seiten erhöht werden kann. Außerdem wird eine Möglichkeit zur Optimierung des Speicherplatzverbrauchs durch Präfix-Komprimierung und geeignete Kodierung der DeweyIDs gezeigt.

Kapitel 5 fasst die Ergebnisse zusammen und gibt einen Ausblick auf zukünftige Arbeiten.

# Kapitel 2 Der XML Transaction Coordinator

Ausgangspunkt dieser Diplomarbeit ist der *XML Transaction Coordinator* (XTC), ein prototypisches natives XML-Datenbanksystem, das implementiert wird, um Forschungsergebnisse in der XML-Verarbeitung mit einer realen Implementierung zu testen. XTC bietet die Möglichkeit, XML-Dokumente zu speichern, auf diese mit *Document-Object-Model* (DOM) [17], *Simple Api for XML* (SAX) [18] und *XML Query Language* (XQuery) [19] zuzugreifen und zu modifizieren ([1]).

Beim Zugriff mit einem SAX Parser wird das Dokument sequentiell verarbeitet und je nach gefundener Komponente (Element, Attribut, Text, Kommentar usw.) eine Event-Handler-Funktion aufgerufen, die von der Anwendung implementiert wird. Dabei werden keine Strukturinformationen des XML-Dokuments gespeichert, was auf der einen Seite den Vorteil hat, dass die Verarbeitung sehr schnell erfolgt und der Speicherverbrauch gering bleibt. Dadurch ist es aber auf der anderen Seite nicht möglich, beliebig auf vergangene Komponenten zuzugreifen. Die dazu notwendige Unterstützung muss durch das Anwendungsprogramm implementiert werden, indem eventuell nochmals benötigte Teile des Dokuments gespeichert werden. Nicht benötigte Teile können verworfen werden. SAX eignet sich damit gut für die sequentielle Verarbeitung von XML-Dokumenten.

Bei DOM hingegen wird ein XML-Dokument in einer Knotenhierarchie dargestellt, dem so genannten DOM-Baum. Dieser Aufbau ermöglicht den navigierenden Zugriff über Funktionen wie beispielsweise *getElementByID()*, *getFirstChild()*, *getNextChild()*, *getLastChild()* oder *setAttribute()*. DOM ist somit zum navigationsbasierten Zugriff auf einzelne Knoten und deren Modifikation geeignet.

XQuery ist eine deklarative Anfragesprache für XML-Dokumente, mit der Elemente und Attribute aus XML-Dokumenten abgefragt werden können. Dabei kann sie das Ergebnis in einem gewünschten Format (z. B. wieder XML) liefern. XQuery nimmt dabei eine ähnliche Rolle für XML-Datenbanksysteme ein wie SQL für relationale Systeme.

Dieses Kapitel gibt zunächst einen Überblick über die Systemarchitektur des XTC-Servers. Daraufhin wird in Abschnitt 2.2 das taDOM-Transaktionsmodell definiert. Es besteht aus dem taDOM-Datenmodell, mit dem die Daten eines XML-Dokuments repräsentiert werden, und den darauf definierten Operationen. Die Knoten des taDOM-Baums werden mit DeweyIDs adressiert, die sich sehr gut eignen, da mit ihnen auch nach dem Einfügen und Löschen von Teilbäumen Neunummerierungen verhindert werden können. Nachdem diese

eingeführt wurden, werden Alternativen genannt, die sich jedoch aus verschiedenen Gründen nicht zur Knotennummerierung im XTC-Server eignen. In Abschnitt 2.3 wird schließlich das Speicherungsmodell erklärt, das beschreibt, wie die Daten des taDOM-Datenmodells auf dem Festspeicher abgelegt werden.

## 2.1 Systemarchitektur

Der XTC-Server ist wie das Fünf-Schichten-Architekturmodell für Datenbanksysteme [2] in fünf aufeinander aufbauende Schichten aufgeteilt (siehe Abbildung 2.1). Über den fünf Schichten liegen die Schnittstellendienste, die den Zugriff der Anwendungsprogramme auf die XML-Daten bereitstellen, der über das FTP- und HTTP-Protokoll erfolgt. Mit einem Treiberpaket können die Anwendungsprogramme DOM-, SAX- und XQuery-Befehle ausführen.

Innerhalb des Modells bilden die Dateidienste die unterste Schicht. Mit ihnen werden Datenseiten in Containerdateien gespeichert und gelesen. Jede Containerdatei besteht aus gleich großen Blöcken, wobei die Größe der Blöcke einer Containerdatei frei konfiguriert werden kann. In den Dateidiensten gibt es für jede Containerdatei einen E/A-Manager, der die Ein- und Ausgabe vornimmt. Die Containerdateien werden bei Bedarf, z. B. wenn die initiale Größe der Installation nicht ausreicht, automatisch erweitert.

In der nächst höheren Schicht, der Propagierungsschicht, wird jedem E/A-Manager ein Puffermanager zugeordnet. Die Aufgabe eines Puffermanagers ist es, Seiten für die über ihm liegende Schicht bereitzustellen und dabei die Anzahl der E/A-Operationen zu den Containerdateien durch Pufferung der Seiten möglichst gering zu halten, da jeder Zugriff eine mechanische Positionierung des Schreib-/Lesekopfes bedeutet und folglich solche Operationen teuer sind. Die Dienste, die auf Seiten zugreifen wollen, teilen dem Puffermanager die gewünschte Seitennummer mit. Dieser prüft daraufhin, ob die Seite bereits im Puffer vorhanden ist oder ob sie von einer Containerdatei mit Hilfe eines E/A-Managers eingelesen werden muss. Wenn die Seite in den Puffer geladen ist, teilt der Puffermanager dem anfragenden Dienst die Pufferposition mit, an der sich die gewünschte Seite befindet. Sobald der Dienst die Seite nicht mehr benötigt, teilt er dies dem Puffermanager durch eine *unfix*-Operation mit. Damit kann die Pufferposition für andere Seiten freigegeben werden. Als Verdrängungsstrategie wird zur Zeit die LRD-V2-Strategie verwendet ([2]).

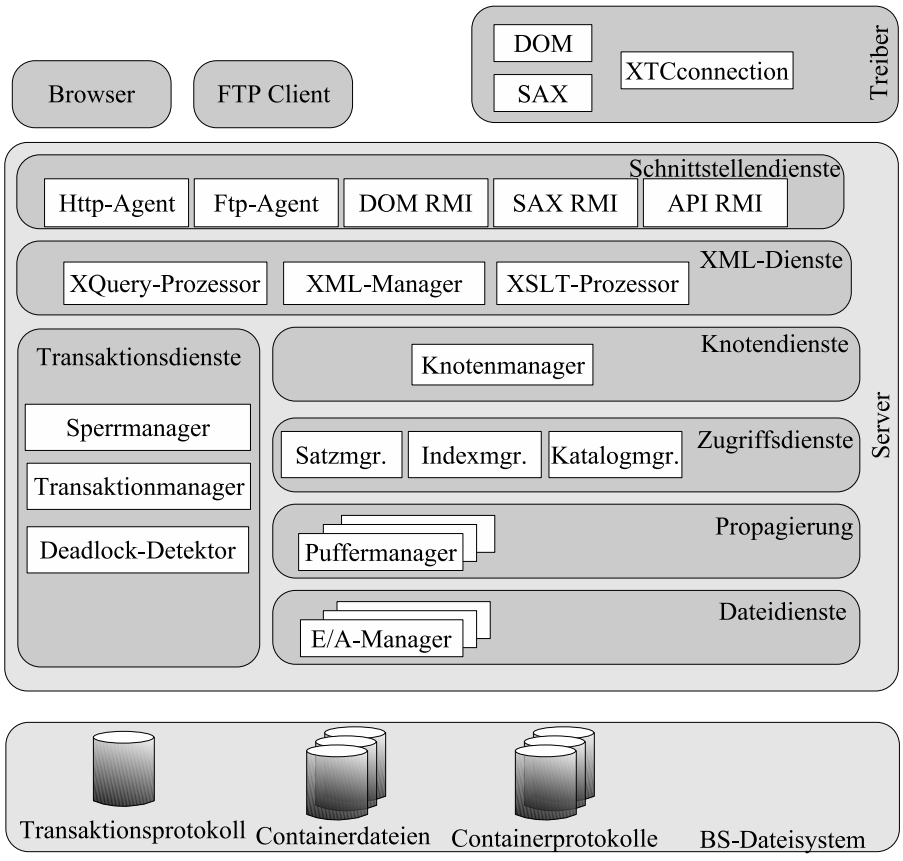
Die Zugriffsdienste befinden sich auf der dritten Ebene des Modells. Sie bilden Elemente, Attribute und Texte auf Datenseiten ab. Diese Schicht besteht aus den drei Komponenten Indexmanager, Satzmanager und Katalogmanager. Der Indexmanager stellt Indexstrukturen wie Listen und B-Bäume zur Verfügung, um Schlüssel/Wert-Paare zu speichern. Ein B*-Baum [5] ist nicht direkt implementiert, sondern wird durch eine indexierte Liste realisiert. Dabei speichert der B-Baum jeweils den ersten Schlüssel jeder Seite der Liste, wodurch es möglich ist, in der Liste die Seite direkt aufzufinden, in der sich der zu suchende Schlüssel befinden muss, wenn er existiert. Der Satzmanager bildet das XML-Dokument bzw. die XML-Knoten auf physische Datensätze ab und speichert diese in den Indexstrukturen in Dokumentenordnung (left-most depth-first) ab. Jeder dieser Knoten erhält eine eindeutige ID

(siehe Kapitel 2.2.3), aus der die Ebene des Knotens feststellbar ist. Mit Hilfe der Dokumentenordnung und der Knotenebene rekonstruiert der Satzmanager das XML-Dokument und nimmt außerdem Modifikationen auf diesen Strukturen vor. Der Katalogmanager verwaltet die Metadaten des Datenbanksystems.

In der Schicht über den Zugriffsdiensten befindet sich der Knotenmanager, der die Umwandlung der internen Speicherungsstruktur auf deren externen Repräsentation durchführt und Operationen zur Navigation auf Knoten und das Einfügen neuer Knoten bietet. Die meisten DOM-Operationen können durch Aufruf äquivalenter Operationen auf dem Knotenmanager ausgeführt werden. Diese Operationen werden transaktionsisoliert durchgeführt, wozu Sperren beim Sperrmanager angefordert und wieder frei gegeben werden.

Die mengenorientierten XML-Dienste bilden die oberste Schicht der Systemarchitektur. Sie erlauben beispielsweise XQuery-Operationen und die Auswertung von XPath-Ausdrücken. Der XML-Manager stellt weiterhin ein virtuelles Dateisystem mit Operationen zum Anlegen, Löschen oder Umbenennen von Verzeichnissen, XML-Dokumenten und beliebigen binären Dateien (BLOBs) zur Verfügung. Mit den Schnittstellendiensten kann der Zugriff auch über das HTTP- oder FTP-Protokoll erfolgen.

**Abbildung 2.1** XTC-Systemarchitektur



## 2.2 Das taDOM-Transaktionsmodell

Damit Transaktionen in einem XML-Datenbanksystem Änderungen auf Knotenebene transaktionsisoliert vornehmen können, ist ein geeignetes Datenmodell zwingend erforderlich, auf dessen Grundlage die Datenhaltung und Anfrageverarbeitung stattfindet. Dieses Datenmodell wird in Abschnitt 2.2.1 eingeführt. Damit die XML-Dienste auf diesem Datenmodell operieren können, werden eine Reihe von Basisoperationen bereitgestellt, mit denen die XML-Schnittstellen auf dieses Grundgerüst abgebildet werden können. Die Basisoperationen, welche sich im Wesentlichen an den standardisierten DOM-Operationen orientieren, werden in Abschnitt 2.2.2 aufgelistet. Zur Anwendung der Operationen auf dem Datenmodell ist eine Knotennummerierung unverzichtbar, da sie in den Operationen zur Adressierung dient. Diese Knotennummerierung erfolgt mit dem Konzept der DeweyIDs, welche in Abschnitt 2.2.3 vorgestellt werden.

### 2.2.1 Das taDOM-Datenmodell

Zur Speicherung der XML-Daten wurde das *transaction document object model* (taDOM) entwickelt, ein DOM-ähnliches Modell, das an die Transaktionsverarbeitung angepasst wurde [3]. Während bei dem DOM-Modell nur Elementknoten, Textknoten und Attribute existieren, werden bei taDOM noch zwei weitere Knotentypen eingeführt: Attributwurzel und String-Knoten. Im taDOM-Modell werden Attribute nicht wie im DOM-Datenmodell direkt an den Elementknoten gehängt, sondern an eine am Elementknoten angehängte Attributwurzel. Dadurch ist es möglich, alle Attribute durch das alleinige Sperren der Attributwurzel zu isolieren. Der String-Knoten im taDOM-Modell enthält den Wert von Attributen und Textknoten, wodurch es möglich ist, auf Knoten unabhängig von ihrem Wert zuzugreifen. Das ermöglicht beispielsweise, dass eine Transaktion auf den Wert eines Elementknotens zugreift und diesen verändert, eine andere Transaktion jedoch gleichzeitig den Elementnamen ausliest. taDOM ist aber nur die Repräsentation eines XML-Dokuments im Arbeitsspeicher. In den Containerdateien werden nur die DOM-Knoten gespeichert, aber mit den DeweyIDs der taDOM-Repräsentation. Abbildung 2.3 zeigt ein Beispiel eines taDOM-Baums, der das XML-Dokument aus Abbildung 2.2 darstellt.

### 2.2.2 Operationen auf taDOM-Bäumen

Auf dem taDOM-Baum sind 19 Basisoperationen definiert, mit denen Knoten gelesen und geändert oder strukturelle Änderungen vorgenommen werden können. Knoten, auf denen diese Operationen angewendet werden, werden mittels DeweyIDs adressiert, die in Kapitel 2.2.3 vorgestellt werden.

#### Basisoperationen zur Navigation

- *getNode(deweyID)*
  Liefert den Knoten mit der angegebenen DeweyID oder *null*, wenn der Knoten nicht existiert.

**Abbildung 2.2** Beispiel eines XML-Dokuments

```
<bib>
     <book year="1994" id="1">
        <title>TCP/IP Illustrated</title>
        <author>
           <last>Stevens</last>
           <first>W.</first>
        </author>
        <price> 65.95</price>
     </book>
     <book year="2000" id="2">
        <title>Data on the Web</title>
        <author>
           <last>Abiteboul</last>
           <first>Serge</first>
        </author>
        <author>
           <last>Buneman</last>
           <first>Peter</first>
        </author>
        <author>
           <last>Suciu</last>
           <first>Dan</first>
        </author>
        <price>39.95</price>
     </book>
     <book year="1999" id="3">
        <title>The Economics of...</title>
        <editor>
           <last>Gerbarg</last>
           <first>Darcy</first>
           <affiliation>CITI</affiliation>
        </editor>
        <price>129.95</price>
     </book>
</bib>
```

- *getParentNode(node)*
  Liefert den Elternknoten des übergebenen Kontextknotens.
- *getPrevSibling(node)*
  Liefert den vorherigen Geschwisterknoten des übergebenen Kontextknotens.
- *getNextSibling(node)*
  Liefert den nächsten Geschwisterknoten des übergebenen Kontextknotens.
- *getFirstChild(node)*
  Liefert den ersten Kindknoten des übergebenen Kontextknotens.
- *getLastChild(node)*
  Liefert den letzten Kindknoten des übergebenen Kontextknotens.

- *getChildNode(node)*
  Liefert eine Liste aller Kinder des übergebenen Kontextknotens.
- *getFragmentNodes(node)*
  Liefert in einer Liste den Kontextknoten und alle seine Nachfahren in Dokumentenordnung.
- *getAttribute(node, attributeName)*
  Liefert für den übergebenen Kontextknoten den Knoten des Attributs mit dem angegebenen Namen.
- *getAttributes(node)*
  Liefert für den übergebenen Kontextknoten eine Liste aller seiner Attributknoten.

#### Basisoperationen zum Auslesen und Ändern von Knotenwerten

- *setAttribute(node, attributeName, attributeValue)*
  Setzt bei dem übergebenen Kontextknoten für das Attribut mit dem angegebenen Namen den spezifizierten Wert. Existiert das Attribut noch nicht, wird es neu angelegt.
- *getValue(node)* bzw. *setValue(node, value)*
  Liefert bzw. setzt den Wert des übergebenen Kontextknotens.
- *renameAttribute(node, attributeName, newAttributeName)*
  Benennt für den übergebenen Kontextknoten den Namen des angegebenen Attributs in den spezifizierten neuen Attributnamen um.

#### Basisoperationen zur strukturellen Modifikation

- *appendChild(node, newNode)* bzw. *prependChild(node, newNode)*
  Fügt den spezifizierten neuen Knoten als neuen letzten bzw. neuen ersten Kindknoten des übergebenen Kontextknotens ein.
- *insertBefore(node, newNode)* bzw. *insertAfter(node, newNode)*
  Fügt den spezifizierten neuen Knoten vor bzw. hinter dem übergebenen Kontextknoten als neuen vorherigen bzw. nächsten Geschwisterknoten ein.
- *deleteNode(node)*
  Löscht den übergebenen Kontextknoten und alle seine Nachfahren.

### 2.2.3 Adressierung mit DeweyIDs

Jeder Knoten im taDOM-Baum muss anhand einer ID eindeutig identifiziert werden können, um einen schnellen und direkten Zugriff über einen Index zu ermöglichen. Damit ist es beispielsweise möglich, in einem Index die IDs aller Knoten mit einem bestimmten Elementnamen zu speichern und diesen Index bei der DOM-Funktion *getElementsByName()* zu nutzen, um diese Knoten aufzufinden. Ohne derartigen Index müsste das gesamte Dokument nach den Knoten durchsucht werden. Die IDs dienen außerdem dazu, ein hierarchisches Sperrverfahren zu implementieren. Deshalb muss es möglich sein, anhand der ID eines Knotens die ID des Elternknotens und weiterer Vorfahren zu bestimmen, um festzustellen,

ob ein Vorfahre eine Sperre hält, die den Kontextknoten betrifft. Diese Identifizierung sollte allein anhand der ID möglich sein, um keinen Zugriff auf Datenseiten zu benötigen, damit eine effiziente Sperrverwaltung sichergestellt werden kann. Ausführliche Erläuterungen des hierarchischen Sperrverfahrens im XTC-Server stellt Literatur [3, 7, 8] zur Verfügung.

Bei der Speicherung eines XML-Dokuments wird jedem Knoten eine DeweyID zugewiesen. Sie definiert eine Ordnung auf den Knoten, mit der man feststellen kann, auf welcher Ebene sie sich befinden. Mit Hilfe der DeweyIDs ist es ebenfalls möglich, zwei Knoten zu vergleichen, um ihre relative Zugehörigkeit (Geschwister, Eltern) zu bestimmen. Eine Ebene drückt die Länge des Pfads eines Knotens zur Dokumentwurzel aus. Damit erhält die Dokumentwurzel die Ebene 0. Ein neuer Elementknoten bekommt die Ebene des Elternknotens erhöht um 1. Dadurch liegen Geschwisterknoten auf gleicher Ebene.

Ein Beispiel für eine DeweyID ist 1.3.5.4.3. Eine DeweyID besteht aus so genannten Divisions (in diesem Beispiel fünf), die durch Punkte getrennt dargestellt werden. Der Wurzelknoten des Dokuments bekommt immer die DeweyID 1. Die Kinder erhalten die DeweyID des Elternknotens erweitert um eine Division (z. B. 1.3 für das erste Kind, 1.5 für das zweite Kind usw.). Dabei werden die zusätzlichen Divisions aufsteigend vom ersten Knoten vergeben. Mit dem Vergleich zweier DeweyIDs kann man somit feststellen, ob ein Knoten vor oder hinter dem anderen liegt. Die Vergabe der Division-Werte wird durch den Parameter *Distance* beeinflusst [4]. Der *Distance*-Wert beschreibt, wie viel Platz zwischen zwei Knoten freigelassen wird, um später neue Knoten einfügen zu können.

Während Abbildung 2.2 ein Beispiel eines XML-Dokuments darstellt, das aus [3] entnommen wurde, zeigt Abbildung 2.3 den taDOM-Baum mit entsprechenden DeweyIDs. In Kapitel 3 wird detailliert die Vergabe der DeweyIDs und der Einfluss des *Distance*-Parameters beschrieben.

### 2.2.4 Alternative Knotennummerierungsverfahren

Aktuelle Forschung beschäftigt sich bisher hauptsächlich mit Knotennummerierungsverfahren, die zur Speicherung von XML-Dokumenten in relationalen Datenbanken eingesetzt werden. Dabei wird ein XML-Knoten durch eine Zeile mit einer Knoten-ID im relationalen Datenbanksystem dargestellt. Da diese Verfahren im XTC-Server zum Einsatz kommen könnten, werden sie im Folgenden mit den DeweyIDs verglichen.

In [15] wird ein Nummerierungsverfahren vorgeschlagen, P-PBiTree, bei dem abhängig von der Anzahl der Geschwisterknoten für jede Division $m = log_2 k$ Bits (k=Anzahl der Geschwisterknoten) verwendet werden. Es ist ebenfalls präfixbasiert, das heißt, die ID des Elternknotens wird in der ID des Kindknotens mit aufgenommen. Allerdings ist es bei diesem Verfahren trotzdem nicht möglich, allein anhand der ID den Elternknoten zu ermitteln, da nur die Länge der vollständigen ID gespeichert wird und nicht bekannt ist, aus wie vielen Bits die letzte Ebene besteht. Diese Information geht zugunsten eines besseren Speicherplatzverbrauches verloren. Da diese Eigenschaft im XTC-Server zum Sperren erforderlich ist, scheidet dieses Verfahren als Einsatzmöglichkeit aus.

In [16] wird das *Dynamic Level Number Scheme* beschrieben, das vom Grundprinzip ähn-

lich funktioniert, wie die im XTC-Server verwendeten DeweyIDs. Es verwendet allerdings eine andere Kodierung und schlägt ein anderes Prinzip vor, wie Knoten ohne die Notwendigkeit einer Neunummerierung eingefügt werden können. In der Grundversion *SIMPLE DOM DLN* muss das DOM-Modell bekannt sein, um zu wissen, wie viele Geschwisterknoten auf einer Ebene zu kodieren sind. Abhängig von dieser Anzahl wird für alle Kindknoten die gleiche Anzahl an Bits verwendet, um die nächste Ebene zu kodieren. Es existieren zwei Trennzeichen: „.“ und „/“. Der Punkt wird verwendet, um Ebenen voneinander zu trennen. Der Slash wird erst beim Einfügen neuer Knoten benötigt. Folgendes Beispiel verdeutlicht das Einfügen neuer Knoten: Soll zwischen den IDs 1.5 und 1.6 eine neue ID eingefügt werden, wird der Slash verwendet, und die ID lautet 1.5/1. Der Slash zeigt also an, dass die nächste Zahl noch zur gleichen Ebene gehört. Auch mehrfaches Verwenden des Slashs ist möglich, so dass zwischen 1.5 und 1.5/1 die 1.5/0/1 eingefügt werden kann.

In der Bit-Darstellung wird der Punkt durch ein 0-Bit und der Slash durch ein 1-Bit kodiert. Für den Wert einer Ebene werden wie bei P-PBiTree $m = log_2 k$ Bits (k=Anzahl der Geschwisterknoten) verwendet. 1.5/1 wird, wenn die erste Ebene ein Bit und die zweite Ebene drei Bits verwendet, mit 1 0 101 1 001 kodiert (der Unterwert einer Ebene wird mit der gleichen Anzahl Bits kodiert wie der Wert). Allerdings gibt der Aufsatz keine Auskunft darüber, wie die Länge der Ebenen gespeichert wird, mit Ausnahme des Hinweises, diese in Metadaten abzulegen. Da jedoch zumindest für jeden inneren Knoten gespeichert werden muss, wie viele Bits die Kinder dieses Knotens verwenden, wird hierfür zusätzlicher Speicherplatz gebraucht. Auch müssen bei jedem neuen Knoten die Metadaten aktualisiert werden.

Ein verwandtes Verfahren, das ohne Metadaten auskommt, ist *FIXED DLN*. Hierbei muss ebenfalls das DOM-Modell bekannt sein. Allerdings ist die Anzahl der Bits für alle Ebenen für jeden Knoten identisch, z. B. vier Bits. Falls die vier Bits nicht ausreichen, alle Geschwisterknoten zu nummerieren (bei über 15 Knoten), wird mit einem „/“ eine Unterebene begonnen. Eine Hauptebene und eine Unterebene reichen aus, um 225 Knoten auf einer Ebene zu nummerieren, die in der Dezimalschreibweise wie folgt aussehen: 1/1, 1/2, 1/3, ..., 1/15, 2/1, 2/2, 2/3, ..., 2/15, 3/1, ..., 3/15, ..., 15/1, ..., 15/15. Bei mehr als 225 Knoten (und maximal 3375) wird von Beginn an die Hauptebene mit zwei Unterebenen genutzt. Der erste Knoten lautet 1/1/1 und der letzte Knoten 15/15/15. Der Vorteil dieses Verfahrens liegt darin, dass sowohl kleine als auch große Fanouts sehr effizient kodiert werden. Nachteilig ist jedoch, dass ohne das Wissen über die Anzahl der Geschwisterknoten dieses Verfahren sehr ineffektiv wird. Wenn initial auf Ebene 1 nur 15 Knoten gespeichert werden, haben diese die IDs von 1.1 bis 1.15. Fügt man dahinter aber nacheinander weitere Knoten ein, werden für 15 weitere Knoten weitere vier Bits (mit dem „/“-Bit schließlich fünf Bits) nötig. Die nachfolgend eingefügten Knoten hätten die IDs 1.15/1 bis 1.15/15. Der 20000. Knoten würde letztendlich über 6.500 Bits benötigen, was absolut inakzeptabel ist. Bei solchen unerwarteten Einfügesequenzen skaliert das Verfahren offensichtlich sehr schlecht. Dieses Verfahren eignet sich demzufolge nur, wenn keine großen Änderungen auf dem Dokument zu erwarten sind.

Als weitere Alternative wird in dem Aufsatz das Verfahren *STREAMING DLN* erläutert, welches ohne das DOM-Modell auskommt und auch keine Metadaten verwendet. Auch hier

wird für jede Ebene die gleiche Anzahl an Bits verwendet. Die Vergabe der Divisions zeigt Tabelle 2.1, die für jede Ebene vier Bits verwendet.

**Tabelle 2.1** Streaming DLN Kodierungstabelle

| Unterebenen | Bitmuster | Anzahl der IDs |
|---|---|---|
| 0 | 0XXX | 1 - 7 |
| 1 | 10XX 1 XXXX | 8 - 71 |
| 2 | 110X 1 XXXX 1 XXXX | 72 - 583 |
| 3 | 1110 1 XXXX 1 XXXX 1 XXXX | 584 - 4679 |

Übersetzt in die Dezimaldarstellung werden die IDs auf Ebene 1 also folgendermaßen vergeben: 1.1, 1.2, ..., 1.7, 1.7/0, 1.7/1, ..., 1.7/15, 1.8/0, 1.8/1, ..., 1.8/15, 1.9/0, ..., 1.11/15, 1.12/0/0, ...

Dieses Nummerierungsverfahren skaliert zwar besser als FIXED DLN, weist aber Mängel bei der Kodierung auf, da bei jeder weiteren verwendeten Unterebene ein zusätzliches Bit „verschenkt“ werden muss. Wie in Kapitel 3.2 zu sehen sein wird, ist das bei der Kodierung der DeweyIDs nicht der Fall, da sie eine effizientere Kodierung verwenden.

### 2.2.5 Transaktionsisolation mit taDOM2, taDOM2+, taDOM3, taDOM3+

Wenn eine Transaktion Änderungen mit den in Kapitel 2.2.2 aufgelisteten Operationen auf einem taDOM-Baum ausführt, müssen Sperren angefordert werden, damit sich nebenläufige Transaktionen nicht gegenseitig stören. Da ein XML-Dokument hierarchisch aufgebaut ist, bietet sich multigranulares Sperren an, ähnlich wie es in relationalen Datenbanken mit Datensätzen, Tabellen und Segmenten erfolgt. Um nebenläufige Transaktionen zu unterstützen, werden in [8] die vier Protokolle taDOM2, taDOM2+, taDOM3 und taDOM3+ vorgestellt. Mit diesen Sperrprotokollen können die Operationen auf dem taDOM-Datenmodell transaktionsisoliert ausgeführt werden. Da DOM, SAX und XQuery-Operationen auf diesen Operationen aufbauen, kann Transaktionssicherheit gewährleistet werden.

Diese Sperrprotokolle werden jedoch nicht weiter in dieser Arbeit behandelt.

## 2.3 Speicherungsmodell

Das taDOM-Datenmodell muss in geeigneter Weise in Datenseiten zerlegt und gespeichert werden. Dazu werden die Knoten des Datenmodells in Datensätze kodiert und in verketteten Seiten zusammen mit ihrer DeweyID in Dokumentenordnung im *Dokumentcontainer* abgelegt. Die Vergabe der DeweyIDs wird in Kapitel 3 eingehend dargelegt. Abbildung 2.5 zeigt die Speicherungsstruktur des XML-Dokuments aus Abbildung 2.2.

Um einen Knoten schnell auffinden zu können, speichert der so genannte *Dokumentindex* die jeweils erste DeweyID jeder Seite des *Dokumentcontainer*. Dieses Speicherungsverfahren bietet die Möglichkeit, einen bestimmten Knoten zu suchen, indem im *Dokumentindex* der

größte Schlüssel festgestellt wird, der kleiner oder gleich dem zu suchenden Schlüssel ist. Anschließend wird die Seite, auf die er verweist, nach dem gefragten Knoten durchsucht. Die DeweyIDs werden dazu in eine Byte-Repräsentation kodiert, wie in Kapitel 3.2 beschrieben wird.

Die Datensätze können auf vier verschiedene Arten gespeichert werden:

- *inlined (materialized)*
  *Inlined* bedeutet, dass die Daten materialisiert, also direkt im Datensatz gespeichert werden.
- *distributed (referenced)*
  Ein Datensatz, der zu groß für eine Seite ist, wird über mehrere Seiten verteilt gespeichert.
- *vocabulary*
  *Vocabulary* bedeutet, dass nur eine Referenz zu einem Element in einem Vokabular gespeichert wird. Das Vokabular ist vor allem bei Elementnamen und Attributnamen vorteilhaft, da sie überlicherweise sehr oft im Dokument vorkommen und somit nur einmal gespeichert werden müssen.
- *linked*
  Mit *linked* wird eine DeweyID eines bereits existierenden Knotens gespeichert. Wenn beispielsweise das Ergebnis einer XQuery-Anfrage materialisiert wird, besteht so die Möglichkeit, Verweise auf real existierende Knoten zu speichern. So können, wenn gewünscht, Änderungen auf dem Ergebnisdokument an das Datenbanksystem propagiert werden.

Abbildung 2.4 zeigt das detaillierte Datensatzformat. Das erste Byte eines Datensatzes beschreibt in den ersten zwei Bits, auf welche Art der Datensatz (inlined, distributed, vocabulary, linked) gespeichert wird. In den letzten sechs Bits wird die Art des Knotens (Elementknoten, Attributknoten, Attributwurzelknoten, Textknoten oder String-Knoten) gespeichert. Die weiteren Bytes sind abhängig von der Art des Datensatzes. Bei *inlined* werden die Datensatzbytes angehängt. Bei *distributed* wird in vier Bytes die Seite gespeichert, in welcher der Datensatz beginnt. Von dieser Seite wird bei Bedarf auf die nachfolgende Seite referenziert, welche wieder eine Folgeseite referenzieren kann usw.. Bei *vocabulary*-Datensätzen werden zwei Bytes zur Identifizierung des Vokabulareintrags gespeichert. Bei *linked* folgt auf das erste Byte die DeweyID, auf die dieser Datensatz verlinkt ist, in Byte-Repräsentation.

Neben dem *Dokumentindex* existiert ein Elementindex, wie in Abbildung 2.6 gezeigt. Er verwaltet alle Elementnamen zusammen mit einem *Knotenindex,* der die DeweyIDs der Elementknoten mit dem entsprechenden Elementnamen enthält. Der Elementindex wird beispielsweise bei der Funktion *getElementsByName* eingesetzt, bei der alle Knoten gesucht werden, die einen bestimmten Elementnamen besitzen. Aber auch bei XQuery-Auswertungen kann dieser Index effizient zum Einsatz kommen.

Zur Implementierung des *Dokumentcontainer* mit *Dokumentindex* und des *Knotenindex* wird ein B*-Baum verwendet, dessen Details in Kapitel 4 dargelegt werden.

**Abbildung 2.3** Beispiel eines taDOM-Baums

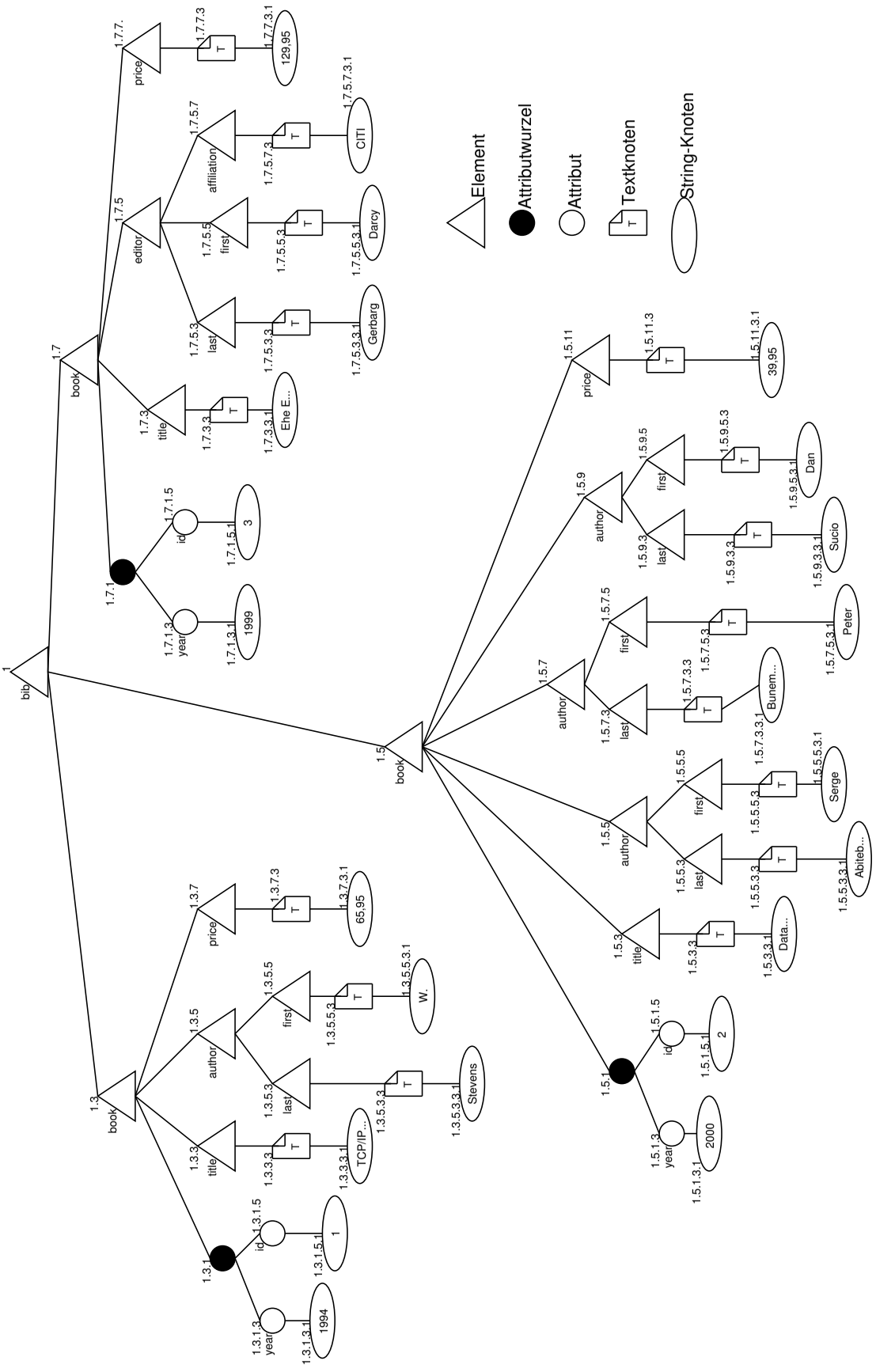


**Abbildung 2.4** Datensatzformate im RecordManager

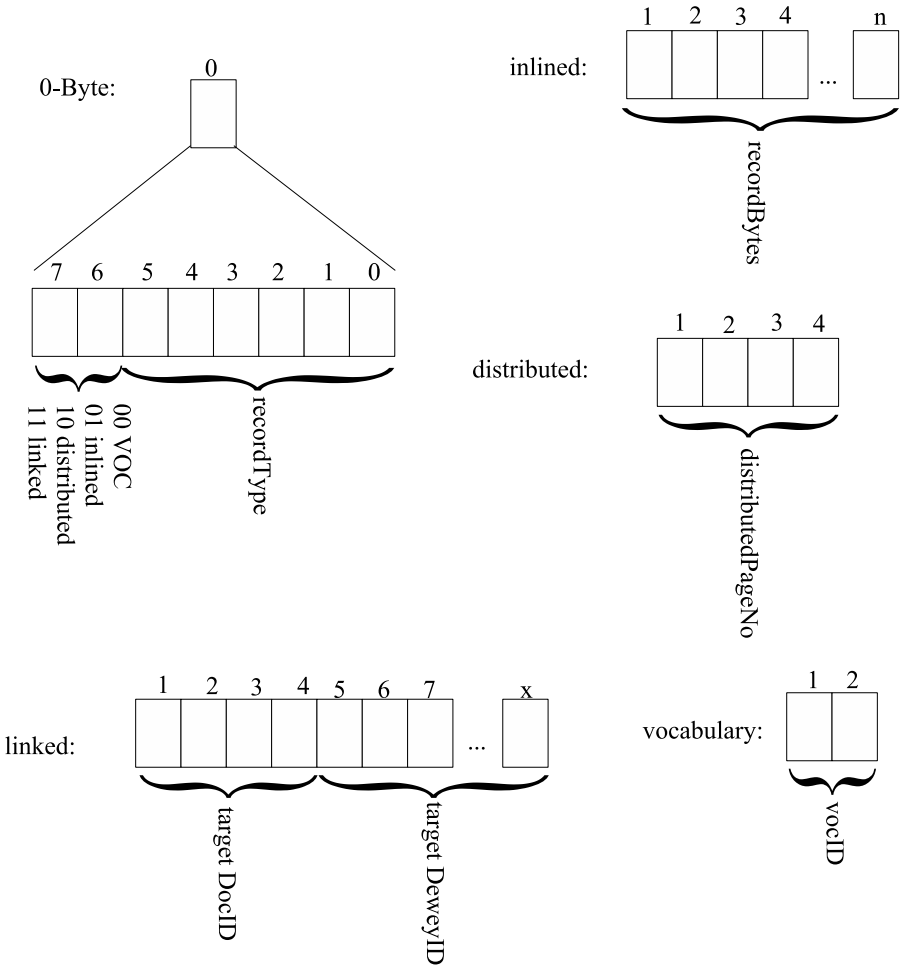



**Abbildung 2.5** Speicherungsstruktur im Dokumentcontainer

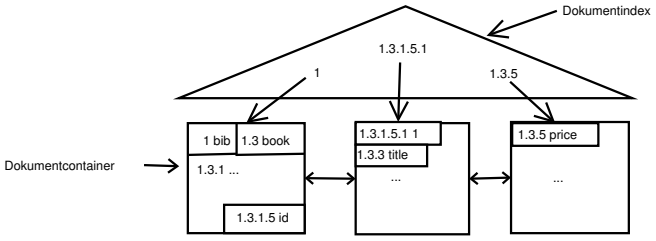
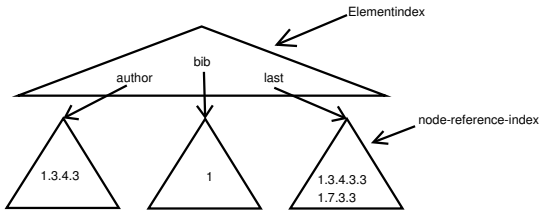



**Abbildung 2.6** Elementindex



# Kapitel 3 DeweyIDs

DeweyIDs basieren auf dem ORDPATHs-Ansatz [6] und dienen in taDOM-Bäumen zur Adressierung von Knoten. Sie bestehen aus mehreren Divisions, die durch Punkte voneinander getrennt sind. Die Divisions bestimmen die Position des Knotens in der Teilbaumebene. Bei den DeweyIDs 1.5.7 und 1.5.9 bestimmt beispielsweise die letzte Division, dass die zweite DeweyID hinter der ersten liegt. In Abschnitt 3.1 wird erläutert, nach welchem Verfahren die DeweyIDs an die taDOM-Knoten vergeben werden. Eine effiziente Kodierung der DeweyIDs in eine Byte-Repräsentation wird in Abschnitt 3.2 eingeführt, deren Speicherplatzverbrauch in 3.3 untersucht wird. Zur Kodierung der DeweyIDs ist eine Kodierungstabelle nötig, deren Aufbau die Effizienz der Kodierung bestimmt. Deshalb werden in 3.4 weitere mögliche Kodierungstabellen vorgestellt. Da Neunummerierungen von Knoten vermieden werden sollten, werden in Abschnitt 3.5 Überlegungen angestellt, wie häufig bei der Knotennummerierung durch DeweyIDs mit der Notwendigkeit einer Neunummerierung zu rechnen ist.

## 3.1 Vergabe von DeweyIDs

Bei der Zuweisung von DeweyIDs unterscheidet man zwischen der initialen Vergabe beim *bulkloading* einerseits und dem nachträglichen Einfügen neuer Knoten andererseits. Ein Parameter *Distance* bestimmt, welche DeweyIDs beim initialen Speichern eines Dokuments vergeben werden bzw. beeinflusst die DeweyIDs, die beim Einfügen von neuen Knoten erzeugt werden. Bei dem kleinsten *Distance*-Wert 2 werden alle ungeraden Divisions verwendet: Wenn an den Knoten mit der DeweyID 1.5 drei Knoten angehängt werden, erhalten diese die DeweyIDs 1.5.3, 1.5.5 und 1.5.7. Die Division 1 wird in diesem Fall nicht verwendet, da sie eine Sonderrolle einnimmt und außer beim Wurzelknoten nur bei Attributenwurzel und Stringknoten verwendet wird.

Bei einem größeren *Distance*-Wert wird eine Lücke zwischen den Knoten gelassen, um leichter neue Knoten zwischen zwei bereits bestehenden Knoten einfügen zu können. Bei Verwendung des *Distance*-Wertes 4 würden die drei Kinder des Knotens mit der DeweyID 1.5 die DeweyIDs 1.5.5, 1.5.9 und 1.5.13 erhalten. Beim initialen Speichern (*bulkloading*) werden nur ungerade Divisions vergeben. Die geraden Divisions werden zur Überlaufbehandlung eingesetzt, wenn zwischen zwei Knoten keine ungeraden Divisions mehr zur Verfügung stehen. Zwischen den Knoten 1.5.5 und 1.5.7 kann so zum Beispiel der Knoten 1.5.6.3 eingefügt werden. Die Divisions 6.3 beschreiben dabei zusammen die Position auf

der letzten Ebene. Die Überlaufbehandlung ermöglicht es theoretisch, unendlich viele Knoten einzufügen, ohne Knoten neu nummerieren zu müssen. In der Praxis ist jedoch eine maximale DeweyID-Länge unumgänglich, da DeweyIDs Seitengrenzen nicht überschreiten dürfen. Die Überlaufbehandlung wird bei der Vergabe neuer DeweyIDs vor dem ersten Knoten (siehe Abschnitt 3.1.2.2) sowie bei Vergabe neuer DeweyIDs zwischen zwei bestehenden Knoten (siehe Abschnitt 3.1.2.3) benötigt. Die Überlaufbehandlung hat aber den Nachteil, dass eine zusätzliche gerade Division benötigt wird, was die Gesamtzahl der Divisions erhöht. Die Benutzung eines *Distance*-Wertes größer 2 ermöglicht es jedoch, neue DeweyIDs einzufügen, ohne dass eine gerade Division benutzt werden muss und verhindert dadurch eine Erhöhung der Gesamtzahl der Divisions. Dabei gilt, je größer der *Distance*-Wert ist, desto mehr Knoten können ohne Überlaufbehandlung eingefügt werden. Allerdings sollte der *Distance*-Wert an die Änderungswahrscheinlichkeit angepasst sein, da sich durch einen hohen *Distance*-Wert auch der Speicherplatz aufgrund der höheren Division-Werte vergrößert. Deshalb kann die Anzahl der Überlaufbehandlungen durch einen hohen *Distance*-Wert verringert werden, er garantiert aber nicht eine Reduzierung der Reorganisationen, die entstehen, sobald eine DeweyID eine einmal definierte maximale Größe überschreitet. Da dies ein lang andauernder Prozess ist, muss die Anzahl der Reorganisationen möglichst gering gehalten werden. Inwieweit verschiedene Werte von *Distance* die Speichergröße beeinflussen und wie viele Knoten im schlechtesten Fall eingefügt werden können, wird anhand von Untersuchungsergebnissen in Kapitel 3.5 diskutiert.

### 3.1.1 Initiale Vergabe der DeweyIDs

Bei der initialen Vergabe der DeweyIDs gelten folgende Regeln:

1. Der Wurzelknoten des Dokuments erhält immer die DeweyID 1.
2. Der erste Knoten einer Ebene bekommt die DeweyID des Elternknotens, erweitert um eine Division mit dem Wert *Distance*+1.
3. Wird ein neuer Knoten hinter dem letzten Knoten einer Ebene erzeugt, bekommt der neue die DeweyID des bis dahin letzten Knotens, wobei der Wert der letzten Division um *Distance* erhöht wird.
4. Ein Elementknoten, der mindestens ein Attribut hat, erhält eine Attributwurzel. Die Attributwurzel bekommt die DeweyID des Elementknotens, erweitert um eine Division mit dem Division-Wert 1.
5. Der Attributknoten erhält die DeweyID der Attributwurzel, erweitert um eine weitere Division. Wenn es sich um den ersten Attributknoten handelt, besitzt diese Division den Wert 3. Ansonsten hat die Division den Wert der letzten Division des letzten Attributknotens addiert um 2. Hier spielt der *Distance*-Wert keine Rolle, da die Reihenfolge der Attribute keinen Einfluss auf den Inhalt des Dokuments hat. Deshalb können neue Attribute immer am Ende der Attributliste eingefügt werden.
6. Bei der Vergabe von DeweyIDs an Textknoten gelten die gleichen Regeln wie für Elementknoten.

7. Der Wert eines Attributs oder eines Textknotens wird in einem String-Knoten gespeichert. Der String-Knoten erhält die DeweyID des Textknotens bzw. Attributknotens, erweitert um eine Division mit dem Division-Wert 1.

In Tabelle 3.1 werden die Regeln auf den Anfang des Dokuments aus Abbildung 2.2 angewendet, wobei ein *Distance*-Wert von 16 angenommen wird.

**Tabelle 3.1** Vergabe der DeweyIDs bei *Distance*-Wert 16

| | Art des Knotens | Regel | DeweyID |
|---|---|---|---|
| bib | Elementknoten | 1 | 1 |
| book | Elementknoten | 2 | 1.17 |
| | Attributwurzel | 4 | 1.17.1 |
| year | Attributknoten | 5 | 1.17.1.3 |
| 1994 | String-Knoten | 7 | 1.17.1.3.1 |
| id | Attributknoten | 5 | 1.17.1.5 |
| 1 | String-Knoten | 7 | 1.17.1.5.1 |
| title | Elementknoten | 2 | 1.17.17 |
| | Textknoten | 6 | 1.17.17.3 |
| TCP/IP... | String-Knoten | 7 | 1.17.17.3.1 |
| author | Elementknoten | 3 | 1.17.33 |
| last | Elementknoten | 2 | 1.17.33.17 |
| | Textknoten | 6 | 1.17.33.17.17 |
| Stevens | String-Knoten | 7 | 1.17.33.17.17.1 |
| first | Elementknoten | 3 | 1.17.33.33 |
| | Textknoten | 6 | 1.17.33.33.17 |
| W. | String-Knoten | 7 | 1.17.33.33.17.1 |
| price | Elementknoten | 3 | 1.17.49 |
| | Textknoten | 6 | 1.17.49.17 |
| 65.95 | String-Knoten | 7 | 1.17.49.17.1 |
| book | Elementknoten | 3 | 1.33 |

### 3.1.2 Einfügen neuer Knoten

Nachdem die Knoten nach den Regeln von Kapitel 3.1.1 erzeugt und im XDBS gespeichert wurden, können im laufenden System neue Knoten eingefügt werden. Dazu müssen für die neuen Knoten DeweyIDs vergeben werden, die wie die bereits bestehenden DeweyIDs die Ebene, die Position in der Teilbaumebene und Art des Knotens wiedergeben. Dabei wird unterschieden, ob der Knoten hinter dem letzten Knoten einer Ebene, vor dem ersten Knoten einer Ebene oder zwischen zwei bestehenden Knoten eingefügt wird. Attributwurzel, Attributknoten und String-Knoten müssen dabei nicht berücksichtigt werden, da diese wie bei der initialen Vergabe behandelt werden.

#### 3.1.2.1 Vergabe neuer DeweyIDs hinter dem letzten Knoten

Die Vergabe neuer DeweyIDs hinter dem letzten Knoten funktioniert sehr ähnlich wie die initiale Vergabe von DeweyIDs. Wenn die letzte Ebene lediglich aus einer Division besteht, wird diese um *Distance* erhöht.

**Beispiel** Der Nachfolger von 1.3.15 bei einem *Distance*-Wert von 16 ist 1.3.31.

Wenn die letzte Ebene aus mehr als einer Division besteht, wird die erste Division dieser Ebene um *Distance*-1 erhöht, so dass eine ungerade Division entsteht.

**Beispiel** Der Nachfolger von 1.3.14.6.5 bei einem *Distance*-Wert von 16 ist 1.3.29.

#### 3.1.2.2 Vergabe neuer DeweyIDs vor dem ersten Knoten

Soll vor dem ersten Knoten einer Ebene ein neuer Knoten eingefügt werden, wird standardmäßig die erste Division der letzten Ebene halbiert und bei Bedarf aufgerundet oder um eins erhöht, um eine ungerade Division zu erhalten. Durch das Halbieren ist sichergestellt, dass vor und nach dem neuen Knoten gleich viel „Platz“ für neue Knoten ist.

**Beispiel** Der Vorgänger von 1.5.9 ist 1.5.5.

Sollten die ersten Divisions der letzten Ebene 2 sein, müssen diese übernommen werden, da ein kleinerer Division-Wert als 2 nicht möglich ist.

**Beispiel** Der Vorgänger von 1.5.2.2.8.9 ist 1.5.2.2.5.

Sollte die erste Division der letzten Ebene 3 sein, werden die Divisions 2.*Distance*+1 angehängt.

**Beispiel** Der Vorgänger von 1.5.3 bei einem *Distance*-Wert von 16 ist 1.5.2.17.

#### 3.1.2.3 Vergabe neuer DeweyIDs zwischen zwei bestehenden Knoten

Wenn zwischen zwei bestehenden Knoten *d1* und *d3* ein neuer Knoten eingefügt werden soll, muss eine neue DeweyID *d2* gefunden werden, die zwischen den DeweyIDs *d1* und *d3* liegt. Da die DeweyIDs auf der gleichen Ebene liegen und benachbart sind, haben sie den gleichen Elternknoten und unterscheiden sich nur auf der letzten Ebene, die aus beliebig vielen geraden Divisions und einer ungeraden Division besteht. Alle gemeinsamen Divisions vor der ersten unterschiedlichen Division sind für die neue DeweyID ebenfalls gleich. Die erste Division, bei der sich die DeweyIDs *d1* und *d3* unterscheiden, bestimmt, welche Division die neue DeweyID erhält, wenn mindestens eine ungerade Division dazwischen passt. Wenn mehr als eine ungerade Division dazwischen passt, wird die mittlere Division bevorzugt, um bei eventuellen späteren Einfügungen noch Divisions zwischen *d1* und *d2* sowie zwischen *d2* und *d3* vergeben zu können, ohne zusätzliche neue Divisions einfügen zu müssen.

**Beispiel:** *d1*=1.5.6.7.5, *d3*=1.5.6.7.16.5: Die Divisions 1.5.6.7 sind gleich. Die erste unterschiedliche Division ist 5 bei *d1* und 16 bei *d3*. Es passen die Divisions 7, 9, 11, 13 und 15 dazwischen. Es wird die mittlere Division 11 verwendet. *d2* ist also 1.5.6.7.11.

Wenn nur noch eine gerade Division zwischen die beiden ersten unterschiedlichen Divisions passt, wird diese verwendet und eine weitere Division mit dem Wert *Distance*+1 angehängt.

**Beispiel:** *d1*=1.5.6.7.5, *d3*=1.5.6.7.7, *Distance*=16: Die Divisions 1.5.6.7 sind gleich. Die erste unterschiedliche Division ist 5 bei *d1* und 7 bei *d3*. Es passt nur Division 6 dazwischen. *d2* ist also 1.5.6.7.6.17.

Wenn keine Division mehr dazwischen passt, bedeutet dies, dass die Division der einen DeweyID ungerade (es ist dann die letzte Division) und die der anderen DeweyID gerade

ist. Da eine DeweyID immer mit einer ungeraden Division endet, hat dies zur Folge, dass weitere Divisions folgen müssen.

**Zuerst wird der Fall betrachtet, dass die DeweyID *d3* noch weitere Divisions hat:**

*d2* erhält die übereinstimmenden Divisions von *d1* und *d3,* die erste unterschiedliche Division von *d3,* alle unmittelbar folgenden Divisions mit dem Wert 2 und die erste dann kommende Division mit einem Wert größer als 2, der durch 2 dividiert wird (Ausnahme: Bei einem Division-Wert 3 werden die Divisions 2.1+*Distance* angehängt). Die letzte Division muss einen ungeraden Wert erhalten. Deswegen muss sie evtl. aufgerundet oder um eins erhöht werden. Ein Erhöhen des letzten Division-Wertes stellt sicher, dass nicht der Division-Wert 1 erzeugt wird, der für die Attributwurzel reserviert ist.

**Beispiel:** *d1*=1.5.6.7.5, *d3*=1.5.6.7.6.2.2.13: Die Divisions 1.5.6.7 sind gleich. Die erste unterschiedliche Division von *d3* wird übernommen, die daran anschließenden Divisions mit dem Wert 2 auch. Die dann folgende Division 13 geteilt durch 2 wird als letzter Division-Wert angefügt und aufgerundet, um eine ungerade Division zu erhalten. *d2* ist also 1.5.6.7.6.2.2.7

**Beispiel:** *d1*=1.5.6.7.5, *d3*=1.5.6.7.6.2.2.3, *Distance*=16: Die ersten Divisions werden wie im Beispiel davor erstellt. Statt der 7 wird aber 2.17 ergänzt. *d2* ist also 1.5.6.7.6.2.2.2.17

**Jetzt wird der Fall betrachtet, dass die DeweyID *d1* noch weitere Divisions hat:**

*d2* erhält die übereinstimmenden Divisions von *d1* und *d3*, die erste unterschiedliche Division von *d1* und zusätzlich die nächste Division von *d1* erhöht um *Distance*. Sollte diese Division gerade sein, wird sie um eins dekrementiert. Hier ist ein Dekrementieren möglich, da mindestens der Wert 2 (minimaler *Distance*-Wert) addiert wurde und somit sichergestellt ist, dass eine gültige DeweyID erzeugt wird.

**Beispiel:** *d1*=1.5.4.5, *d3*=1.5.5, *Distance*=16: Die Divisions 1.5 sind gleich. Die erste unterschiedliche Division von *d1* wird übernommen. Die dann folgende Division 5 wird um *Distance* erhöht und angefügt. *d2* ist also 1.5.4.21

Diese Form der Nummerierung hat mehrere Vorteile:

- Zwischen zwei bestehenden DeweyIDs kann fast immer eine neue ID eingefügt werden, ohne die IDs anderer Knoten reorganisieren zu müssen. Eine Neunummerierung ist nur erforderlich, wenn eine DeweyID in ihrer Byte-Repräsentation eine bestimmte Größe übersteigt und somit die maximale Länge zur Speicherung eines Schlüssels im B*-Baum erreicht ist (siehe Kapitel 3.5).

- Die DeweyID des Elternknotens kann sehr einfach ermittelt werden, indem die letzte Division (immer ungerade) und evtl. die dann noch am Ende befindlichen geraden Divisions entfernt werden. Dazu muss nicht auf das gespeicherte Dokument zugegriffen werden (wichtig für eine effiziente hierarchische Sperrverwaltung).

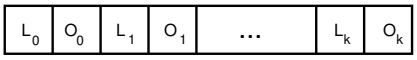
- Durch den Vergleich der Divisions zweier DeweyIDs kann bestimmt werden, welche ID größer oder kleiner ist, d.h., es existiert eine Ordnung auf den IDs, die bei der Speicherung des gesamten XML-Dokuments ausgenutzt werden kann.

- Um zu prüfen, ob eine DeweyID *d1* Vorfahre von DeweyID *d2* ist, muss nur geprüft werden, ob DeweyID *d1* ein Präfix von DeweyID *d2* ist.
- Indem man beim initialen Einspeichern eines XML-Dokuments eine hohe Schrittgröße benutzt (diese ist gegen den erhöhten Speicherplatzbedarf abzuwägen), kann man die Anzahl der Reorganisationen verringern. Dies wird detailliert in Kapitel 3.5 untersucht.

## 3.2 Byte-Repräsentation von DeweyIDs

Eine DeweyID wird in B- und B*-Bäumen als Byte-Folge in Datenseiten gespeichert. Dazu ist eine effiziente Byte-Darstellung nötig, für die jede einzelne Division in einer Bit-Folge kodiert wird. Diese Bit-Folgen ergeben zusammen die DeweyID. In den taDOM-Bäumen des XDBS treten kleine Divisions meist öfter auf als große. Kleine Divisions entstehen dadurch, dass die „ersten“ Kinder eines Knotens kleinere Divisions erhalten als die späteren Kinder und es zwangsläufig immer mehr „erste” Kinder gibt. Diese Tatsache wird in der Byte-Repräsentation berücksichtigt, indem für kleine Division-Werte weniger Bits benötigt werden als für größere Division-Werte. Abbildung 3.1 veranschaulicht die Speicherung einer DeweyID.

**Abbildung 3.1** DeweyID Format




Eine Division besteht in Anlehnung an [6] aus einem $L_i$/$O_i$-Paar. $L_i$ bestimmt dabei die Bit-Anzahl von $O_i$. $O_i$ bestimmt dann den eigentlichen Wert der Division. Tabelle 3.2 zeigt, welche Länge durch $L_i$ definiert wird und welcher Wertebereich $O_i$ bestimmen kann, also welchen Wert die Division mit dem entsprechenden $L_i$ annehmen kann. In der Tabelle wurde der Wertebereich auf *MaxInt* begrenzt.

### 3.2.1 Details über die Speicherung einer Division

$L_i$ ist präfixfrei, d.h., kein $L_i$ hat ein anderes $L_i$ als Präfix. Da beispielsweise $L_i$ 1101 existiert, kann kein anderes $L_i$ mit 1, 11 oder 110 existieren. Man kann somit in einem Bit-String exakt bestimmen, wo $L_i$ endet, wenn bekannt ist, wo es beginnt. Dazu wird das erste Bit in der ersten Spalte der Tabelle 3.2 gesucht. Wenn ein gleicher Wert auftritt, hat man $L_i$ gefunden. Andernfalls werden die ersten zwei Bits in der Tabelle gesucht usw.. Wenn man $L_i$ gefunden hat, weiß man, welche Länge $O_i$ hat und dass es bei dem unmittelbar folgenden Bit beginnt.

Mit Hilfe von Tabelle 3.2 kann man anschließend ablesen, welchen Wert $O_i$ definiert. Die Tabelle, wie sie hier steht, ist nur ein Beispiel. Man kann diese Tabelle auch anders aufbauen, wenn damit eine effizientere Speicherung bei den zu verwaltenden Dokumenten erzielt wird. Kapitel 4.6.3 beschäftigt sich ausführlich mit diesem Thema.

**Tabelle 3.2** $L_i/O_i$-Tabelle

| Bit-String | $L_i$ | $O_i$ Wertebereich |
|---|---|---|
| 0 | 3 | 1-7 |
| 100 | 4 | 8-23 |
| 101 | 6 | 24-87 |
| 1100 | 8 | 88-343 |
| 1101 | 12 | 344-4439 |
| 11100 | 16 | 4440-69975 |
| 11101 | 20 | 69976-1118551 |
| 11110 | 24 | 1118552-17895767 |
| 11111 | 31 | 17895768-2147483647 |

### 3.2.2 Beispiele zur Speicherung einer Division

Um eine Division, z. B. 13, zu kodieren, geht man wie folgt vor: Man schaut in Tabelle 3.2, in welchen Bereich der Wert 13 passt. Das ist in diesem Fall der zweite Wertebereich mit $L_i$ 100. Durch dieses Präfix wird der Wertebereich 8-23 definiert. Außerdem sieht man in der Tabelle, dass $O_i$ aus vier Bits besteht. Wenn $O_i$ 0000 ist, wird der erste Wert des Wertebereichs beschrieben, also 8. Bei 0001 der zweite Wert (also 9) und bei 0010 der dritte Wert (also 10) usw.. 13 ist der sechste Wert, so dass sie mit 0101 kodiert wird. $L_i$ und $O_i$ zusammengesetzt ergeben dann 1000101 zur Kodierung der Division 13.

Einen Sonderfall in der Tabelle bildet der erste Wertebereich. Obwohl man mit drei Bits acht verschiedene Werte darstellen kann, werden lediglich sieben verwendet. Das hat seine Ursache darin, dass die 1 mit 001 und nicht mit 000 kodiert wird. Da in der Byte-Repräsentation der DeweyIDs das letzte Byte mit Nullen aufgefüllt wird, wäre die Division 1 sonst nicht von füllenden Nullen unterscheidbar.

Den umgekehrten Weg, das Dekodieren eines Bit-Strings zu einer Division, soll am Bit-String 101000011 erläutert werden. Zuerst muss $L_i$ bestimmt werden. Die ersten beiden Bits 10 sind in der Tabelle nicht als $L_i$ gespeichert. Bei der Untersuchung der ersten drei Bits stellt man fest, dass 101 als $L_i$ existiert und somit $O_i$ aus sechs Bits bestehen muss, in diesem Beispiel 000011. Der Wertebereich, der von $L_i$ 101 beschrieben wird, ist 24-87. Wenn alle Bits von $O_i$ 0 sind, wird der erste Wert des Wertebereichs beschrieben, wenn alle Bits 1 sind, wird der letzte Wert beschrieben. Daher wird durch 000011 der vierte Wert kodiert, also der Division-Wert 27.

### 3.2.3 Beispiel zur Speicherung einer DeweyID

Das Speichern einer DeweyID erfolgt durch das Zusammensetzen der kodierten Divisions. Da alle DeweyIDs mit der Division 1 beginnen, muss die erste Division nicht explizit gespeichert werden. Wie in Abschnitt 3.2.2 dargelegt wurde, wird Division 27 mit der Bit-Folge 101000011 kodiert. Zum Speichern der DeweyID 1.7.27 benötigt man also nur noch die Kodierung der Division 7, d. h. in diesem Fall die Bits 0111. Durch das Zusammensetzen der

zwei Bit-Folgen entsteht 01111010.00011. Da die Speicherung der Bit-Folge in einer Byte-Folge geschieht, sind zur besseren Übersichtlichkeit Punkte zwischen den einzelnen Bytes eingefügt. Das letzte Byte besteht nur aus fünf Bits, weshalb noch drei Bits aufgefüllt werden müssen. Die endgültige Speicherung sind zwei Bytes die aus folgenden Bits bestehen: 01111010.00011000.

### 3.2.4 Implementierungsdetails

Dieser Abschnitt zeigt, wie die DeweyIDs effizient implementiert werden können, d. h., wie die Kodierung einer DeweyID in eine Byte-Repräsentation und umgekehrt die Umwandlung einer Byte-Repräsentation in eine DeweyID vorgenommen werden kann. Die folgenden Abschnitte legen die dabei benötigten Datenstrukturen und Algorithmen im Detail dar.

#### 3.2.4.1 Umwandlung einer DeweyID in eine Byte-Repräsentation

Zur Implementierung muss es möglich sein, auf die Werte der Tabelle 3.2 zuzugreifen. Folgender Aufbau der Datenstrukturen ermöglicht den Zugriff:

1. ein statisches Array, in dem die Länge $L_i$ gespeichert wird
2. ein Array, in dem der Bit-String, der $L_i$ repräsentiert, durch boolsche Variablen gespeichert wird

Die fehlenden Inhalte der Tabelle 3.2 können durch diese Informationen errechnet werden. Aus Effizienzgründen wird noch ein weiteres Array erstellt, in dem die komplette Länge der Divisions gespeichert wird, um diese schnell ermitteln zu können, sowie ein weiteres statisches Array, in dem die maximalen Werte der Wertebereiche gespeichert werden. Beim Umwandeln einer DeweyID in ihre Byte-Repräsentation kann man mit Hilfe dieser Arrays schnell die benötigten Bytes errechnen und den entsprechenden Speicherplatz in einem Byte-Array initialisieren. Tabelle 3.3 zeigt die gespeicherten Arrays logisch nebeneinander.

**Tabelle 3.3** Tabelle zur Implementierung

| max. Wert | $L_i$ | Bit-String | Bit-Länge |
|---|---|---|---|
| 7 | 3 | 1 | 4 |
| 23 | 4 | 100 | 7 |
| 87 | 6 | 101 | 9 |
| 343 | 8 | 1100 | 12 |
| 4439 | 12 | 1101 | 16 |
| 69975 | 16 | 11100 | 21 |
| 1118551 | 20 | 11101 | 25 |
| 17895767 | 24 | 11110 | 29 |
| 2147483647 | 31 | 11111 | 36 |

Zur Berechnung der Byte-Repräsentation einer DeweyID werden die Bits der einzelnen Divisions nacheinander in ein Byte-Array eingetragen. Dazu stellt man als erstes über den Division-Wert fest, welches $L_i$ den korrekten Wertebereich definiert. Dazu wird der Bit-String der Zeile mit dem kleinsten maximalen Wert gewählt, der dem Division-Wert ent-

spricht. Man speichert nun den Bit-String für die erste Division beginnend bei dem ersten Bit des Byte-Arrays. $O_i$ berechnet man durch Subtraktion des um eins erhöhten maximalen Wertes des nächstkleineren Wertebereichs von dem zu speichernden Division-Wert. Wird der Bit-String der ersten Zeile verwendet, muss nichts subtrahiert werden. Dieser berechnete Wert $O_i$ wird gespeichert, wobei dazu $L_i$ Bits benötigt werden. Das Setzen der entsprechenden Bits kann effizient durch den logischen Bit-Operator *OR* geschehen (128 beim 1. Bit, 64 beim 2. Bit, 32 beim 3. Bit, 16 beim 4. Bit, 8 beim 5. Bit, 4 beim 6. Bit, 2 beim 7. Bit und 1 beim 8. Bit). Durch einen Bit-Index, der auf ein Bit des Byte-Array zeigt, kann schnell festgestellt werden, welches Bit von welchem Byte gesetzt werden muss. Das Speichern der nächsten Division beginnt beim unmittelbar folgenden Bit der vorherigen Division.

Beispiel: Die DeweyID 1.3.11 soll in ihre Byte-Repräsentation umgewandelt werden. Da alle DeweyIDs mit der Division 1 beginnen, fängt die Kodierung bei Division 3 an. Für die Division 3 wird zur Kodierung die erste Zeile aus Tabelle 3.3 verwendet, für Division 11 entsprechend die zweite Zeile. Die Bit-Längen werden addiert, so dass elf Bits zur Speicherung der DeweyID erforderlich sind. Da die DeweyID in einem Byte-Array gespeichert wird, müssen für die benötigten elf Bits zwei Bytes initialisiert werden. Der Bit-Index steht hier auf dem ersten Bit:

```
0 0 0 0 0 0 0 0 . 0 0 0 0 0 0 0 0
↑
```

Als erstes wird der Bit-String 0 für die Division 3 gespeichert. Da das erste Bit 0 ist, muss nur der Bit-Index weiterbewegt werden.

```
0 0 0 0 0 0 0 0 . 0 0 0 0 0 0 0 0
  ↑
```

Die Speicherung des Bit-Strings der ersten Division ist jetzt abgeschlossen. Als nächstes muss $O_i$ errechnet und gespeichert werden. In der ersten Zeile entspricht $O_i$ den zu speichernden Divisions, da 000 aus in Kapitel 3.2.2 beschriebenen Gründen nicht verwendet wird. 3 in drei Bits kodiert ergibt 011. Nach dem Speichern dieser drei Bits hat das Byte-Array den folgenden Zustand:

```
0 0 1 1 0 0 0 0 . 0 0 0 0 0 0 0 0
      ↑
```

Das Speichern des zweiten Division-Wertes 11 geschieht analog. Die 1 des Bit-Strings 100 wird durch OR 8 erreicht, und der Bit-Index wird weitere zwei Mal weitergesetzt.

```
0 0 1 1 1 0 0 0 . 0 0 0 0 0 0 0 0
        ↑
```

Für $O_i$ wird $11-(7+1) = 3$ in vier Bits gespeichert (0011). Das bedeutet, dass der Bit-Index zwei Mal weitergesetzt wird.

$$0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ .\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0$$
$$\uparrow$$

Die beiden 1-Bits werden durch OR 32 und OR 16 gesetzt. Jetzt ist die Byte-Repräsentation vollständig konstruiert.

$$0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ .\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0$$

#### 3.2.4.2 Umwandlung einer Byte-Repräsentation in eine DeweyID

Zum schnellen Umwandeln der Byte-Repräsentation in eine DeweyID werden weitere Hilfsstrukturen verwendet. Als erstes benötigt man ein Array, um den präfixfreien Bit-String bei der Dekodierung einer Byte-Repräsentation zu identifizieren. Dieses Array ist in Tabelle 3.4 dargestellt.

Das Array ist so aufgebaut, dass beim Lesen der einzelnen Bits eines Bit-Strings, abhängig von der aktuellen Position und dem gelesenen Bit, eine neue Position berechnet wird und so festgestellt werden kann, wann ein Bit-String zu Ende gelesen ist. Solange der Bit-String noch nicht fertig gelesen ist, beträgt der Wert auf der aktuellen Position 0. Wenn der komplette Bit-String eingelesen ist, hat die aktuelle Position den gleichen Wert wie das dazugehörige $L_i$, also die Anzahl der Bits, die den Wert aus dem Wertebereich bestimmen. Da die Bit-Strings aus Tabelle 3.2 präfixfrei sind, wird verhindert, dass der Cursor zwischendurch auf eine Position zeigt, in dem das $L_i$ eines anderen Bit-Strings gespeichert ist.

**Tabelle 3.4** Array zur Identifizierung eines Bit-Strings

| Pos | 0 | 1 | ... | 11 | 12 | ... | 27 | 28 | ... | 59 | 60 | 61 | 62 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Wert | 0 | 3 | 0 | 4 | 6 | 0 | 8 | 12 | 0 | 16 | 20 | 24 | 31 |

Um einen Bit-String zu identifizieren, setzt man einen Cursor bei Beginn der Dekodierung auf Position 0. Der Cursor wird dann abhängig von den eingelesenen Bits bewegt:

$$neue\,Position = 2 * aktuelle\,Position + 1 + gelesenes\,Bit \tag{3.1}$$

Sobald die aktuelle Cursor-Position auf ein Element zeigt, dessen Wert ungleich Null ist, wurde ein Bit-String gefunden. In dem Element ist $L_i$ gespeichert, d. h. die Bit-Länge von $O_i$. Im folgenden Schritt werden die nächsten $L_i$ Bits eingelesen und in ihren Dezimalwert umgerechnet. Dieser Wert wird mit dem Wert addiert, der in Tabelle 3.5 an der gleichen Cursor-Position steht, die auch in Tabelle 3.4 verwendet wurde. Tabelle 3.5 enthält den Wert des entsprechenden Wertebereichs, der kodiert wird, wenn alle Bits von $O_i$ 0 sind. Zur Bestimmung des Division-Wertes müssen dieser Wert und $O_i$ addiert werden.

Die erste Division ist jetzt errechnet. Die nächsten Divisions werden nach dem gleichen Verfahren ermittelt.

**Tabelle 3.5** Array zur Berechnung des Division-Wertes

| Pos | 0 | 1 | ... | 11 | 12 | ... | 27 | 28 | ... | 59 | 60 | 61 | 62 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Wert | 0 | 0 | 0 | 8 | 24 | 0 | 88 | 344 | 0 | 4440 | 69976 | 1118552 | 17895768 |

Beispiel: Die Bit-Folge 1001001 soll in eine DeweyID umgewandelt werden. Da die erste Division einer DeweyID immer 1 ist, wird sie nicht in der Byte-Repräsentation explizit gespeichert. Zur Bestimmung der zweiten Division stellt man den Cursor anfangs auf Position 0. Das erste Bit der Bit-Folge ist 1. 2*0+1+1 ist 2. Auf Position 2 in Tabelle 3.4 steht eine 0. Deshalb muss das nächste Bit gelesen werden, welches eine 0 ist. 2*2+1+0 ergibt 5. Auch auf dieser Position befindet sich eine 0. Die nächste Position lautet 11 (2*5+1+0). In dem Array ist auf Position 11 die 4 gespeichert. Diese 4 entspricht der 4 aus der zweiten Zeile von Tabelle 3.2. Somit ist $L_0$ gelesen, und es müssen nun die nächsten vier Bits für $O_0$ gelesen werden: 1001. Im Dezimalsystem ist das 9. Addiert mit dem Wert 8, der in Tabelle 3.5 an Position 11 steht, erhält man nun als zweiten Division-Wert 17. Um den nächsten Division-Wert zu ermitteln, wird die Cursor-Position in Tabelle 3.4 wieder auf 0 gesetzt und nach dem gleichen Verfahren vorgegangen.

Tabelle 3.4 ist an die Kodierungstabelle 3.3 angepasst. Wird eine andere Tabelle verwendet, muss auch das Array (*binaryTreeSearch*) angepasst werden. Um ein passendes Array zu erstellen, muss zuerst die benötigte Größe bestimmt werden. Dazu wird erst die Länge des längsten Bit-Strings ermittelt (*maxBitStringLength*). Anschließend wird die maximale Position errechnet, die bei dieser Länge erreicht werden kann. Sie entsteht, wenn alle Bits 1 sind. Über folgenden Algorithmus, der die Formel 3.1 zur Positionsbestimmung verwendet, wird die maximale Position errechnet:

```
int index = 0;
for (int i=0; i < maxBitStringLength; i++)
{
    index = (2 * index) + 2;
}
```

Der Wert von *index* nach der Ausführung gibt die Größe an, die das Array erhalten muss. Bei der Initialisierung werden alle Werte mit 0 belegt. Im Anschluss wird über eine Iteration über alle Wertebereiche die Länge jedes einzelnen Wertebereichs ($L_i$) eingetragen. Für jeden Wertebereich wird nach der Formel 3.1 die Position bestimmt, an der die Länge des Wertebereichs stehen muss, um dort die Länge (*divisionLength*) einzutragen.

```
for (int i=0; i < AnzahlDerWertebereiche; i++) {
    //beginne bei Bitposition 0
    index=0;
    //bestimme die Indexposition dieses Wertebereichs
    for (int j=0; j < LängeVonWerteBereich[i]; j++) {
```

```
            //bestimme die nächste Position abhängig vom gelesenen Bit
            if (bitStringAsBoolean[i][j]==true) {
                index = (2*index) + 2;
              } else {
                index = (2*index) + 1;
              }
          }
          //trage die Länge für diesen Wertebereich ein
          binaryTreeSearch[index] = divisionLength[i];
      }
```

Um Tabelle 3.5 zu erhalten, trägt man an die gleichen Positionen, die entsprechend in Tabelle 3.4 belegt sind, den Wert ein, den dieser Wertebereich hat, wenn alle Bits 0 sind. Das ist mit Ausnahme des ersten Wertebereichs der kleinste darstellbare Wert dieses Wertebereichs. Da der erste Wertebereich „000...0“ nicht nutzen kann, muss bei ihm vom kleinsten Wert (1) noch 1 subtrahiert werden. Deshalb steht an Byte-Position 1 in Tabelle 3.5 der Wert „0“. Würde der Wertebereich bei „0“ beginnen, müsste an dieser Position „-1“ eingetragen werden.

#### 3.2.4.3 Vergleich zweier DeweyIDs

Es ist sehr oft notwendig, zwei DeweyIDs miteinander zu vergleichen. Beispielsweise startet die Suche nach einer DeweyID am Anfang einer Seite. Der Suchvorgang läuft nun so ab, dass die zu suchende DeweyID so lange mit den DeweyIDs der Seite verglichen wird, bis eine DeweyID gefunden ist, die einen größeren oder gleichen Wert wie die gesuchte DeweyID besitzt. Üblicherweise werden dazu die DeweyIDs aus den Byte-Repräsentationen der Seite erstellt und mit der zu suchenden DeweyID verglichen. Die Vergleichsoperation vergleicht dabei Division für Division. Die erste unterschiedliche Division entscheidet schließlich, welche DeweyID die größere ist.

Der Nachteil dieser Methode liegt darin, dass viele DeweyID-Objekte allein für den Vergleich erstellt werden, danach aber nicht mehr gebraucht werden. Das Erstellen einer DeweyID benötigt jedoch relativ viel Zeit, da ein neues Objekt erstellt und die Byte-Repräsentation in Divisions dekodiert werden muss. Die hier vorgestellte Byte-Repräsentation ermöglicht aber auch einen Vergleich auf Byte-Ebene. Dies ist möglich, da die Werte der Kodierungstabelle die Ordnung in der Bit-Repräsentation beibehalten, da die Bit-Strings aufsteigend nach Wertebereichen angeordnet sind und außerdem innerhalb eines Wertebereichs die Werte aufsteigend kodiert werden. Zwei DeweyIDs, bei denen die erste unterschiedliche Division $i$ im gleichen Wertebereich liegt, unterscheiden sich durch $O_i$. Da wie dargelegt die Wertebereiche aufsteigend mit ihrem Wert kodiert werden, resultiert daraus auch ein größerer Byte-Wert für die größere DeweyID. Wenn bei den beiden DeweyIDs die erste unterschiedliche Division in verschiedenen Wertebereichen liegt, ist die DeweyID, deren Division im größeren Wertebereich liegt, die größere DeweyID. Da die Bit-Strings, die über den Wertebereich entscheiden, aufsteigend kodiert werden, entsteht auch hier bei der größeren DeweyID der größere Byte-Wert.

Der Vergleich zweier DeweyIDs in Byte-Repräsentationen geschieht Byte für Byte. Das erste unterschiedliche Byte bestimmt dabei die größere DeweyID, da an dieser Stelle unterschiedliche Divisions kodiert werden. Dabei müssen die einzelnen Bytes als *unsigned,* also vorzeichenlos betrachtet werden. Die DeweyID, die das größere Byte besitzt, ist auch die größere DeweyID.

Um die Effizienzsteigerung zu zeigen, wurde zwei Millionen Mal die DeweyID „1.15001.17.5.3“ mit der DeweyID „1.15001.17.5.37“ verglichen. Die erste Methode benötigte dazu knapp 12 Sekunden, während der Vergleich auf Byte-Ebene nur 157 Millisekunden benötigte! Das bedeutet eine Geschwindigkeitssteigerung um das 75-fache. Bei einem weiteren Test mit längeren DeweyIDs (Ebene 17) erhöhte sich die Geschwindigkeitssteigerung sogar auf das 200-fache.

## 3.3 Speicherplatzverbrauch

Um die Leistungsfähigkeit der vorgestellten DeweyIDs zu beurteilen, wurden verschiedene XML-Dokumente mit unterschiedlichen strukturellen Eigenschaften hinsichtlich ihres Speicherplatzverbrauchs bezüglich der DeweyIDs untersucht. Dabei handelt es sich um die Dokumente *customer.xml*, *orders.xml* und *lineitem.xml*, die aus dem TPC-H-Benchmark stammen, und inhaltlich eine Kundendatenbank, Bestellungen und bestellte Güter speichern. *ebay.xml* stellt Ebay-Auktionsdaten bereit, *nasa.xml* astronomische Daten und das Dokument *uwm.xml* Kurse einer Universitätswebseite. *mondial*-3.0*.xml* ist eine geographische Datenbank, die Sparten wie z. B. Kontinente, Länder, Regionen, Städte usw. enthält und in diesen viele Attribute aufweist, die Informationen speichern, wie z. B. Name, Einwohnerzahl, Querverweise usw.. *SwissProt.xml* und *psd7003.xml* sind Proteinsequenzen-Datenbanken, und *treebank_e.xml* ist eine verschlüsselte Datenbank englischer Sätze aus dem Wall Street Journal. Außerdem wurde *dblp.xml* untersucht, das bibliographische Informationen aus Aufsätzen und Publikationen der Informatik enthält. Die Dokumente stammen bis auf *dblp.xml*, das von [14] runtergeladen wurde, aus dem XML-Repositorium der Universität Washington [13].

Für die Untersuchungen wurde für jeden Knoten, wie in Kapitel 3.1.1 beschrieben, eine DeweyID erstellt und anschließend der durchschnittliche Speicherplatzverbrauch einer DeweyID in der Byte-Repräsentation bei verschiedenen *Distance*-Werten berechnet. Diese Untersuchung wurde mit *Distance* 2 bis 256 durchgeführt und die Ergebnisse grafisch dargestellt. Bei der Untersuchung fanden jedoch nur Element-, Attribut- und Textknoten Berücksichtigung, da die im taDOM-Modell zusätzlich eingefügten Knoten (String-Knoten, Attributwurzel) nicht auf dem Externspeicher abgelegt werden. Tabelle 3.6 zeigt zusammenfassend die strukturellen Eigenschaften der Dokumente. Zur Berechnung der Ebene erhält der Wurzelknoten die Ebene 0 und ein Kindknoten erhält die Ebene des Vaterknotens erhöht um 1.

Im Folgenden wird zuerst in Unterabschnitt 3.3.1 der Speicherplatzverbrauch im *Dokumentcontainer* (siehe Abbildung 2.5) veranschaulicht, worauf in Unterabschnitt 3.3.2 die

Darlegung des Speicherplatzverbrauchs der Knotenindizes im Elementindex (siehe Abbildung 2.6) erfolgt.

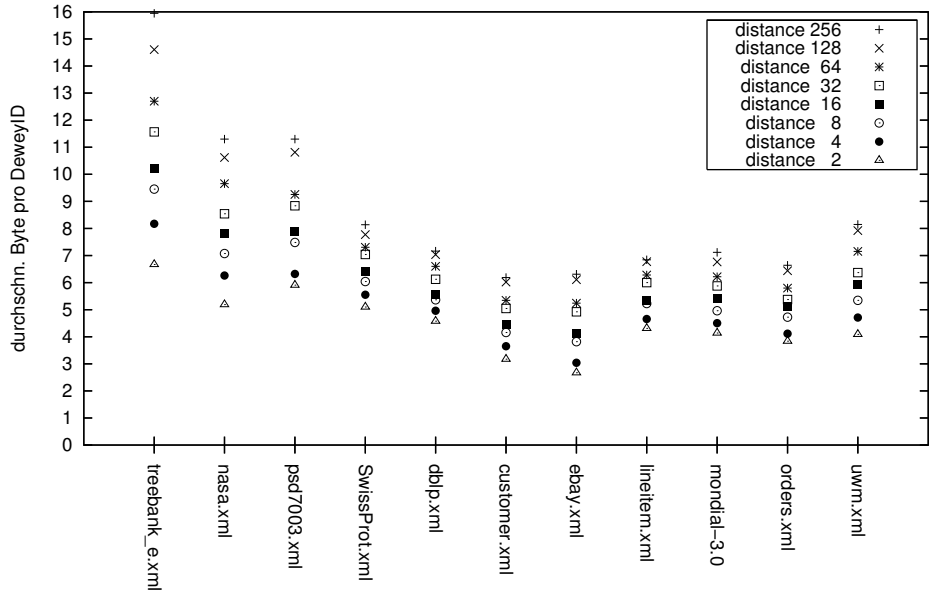
**Tabelle 3.6** XML-Dokumente

| Dateiname | Beschreibung | Größe (Byte) | Anzahl der Element-knoten | Anzahl der Attribute | Anzahl der Text-knoten | max Ebene | ∅ Ebene | max FanOut | ∅ FanOut |
|---|---|---|---|---|---|---|---|---|---|
| treebank_e.xml | Verschlüsselte Datenbank englischer Sätze aus dem Wall Street Journal | 86082517 | 2437666 | 1 | 1391845 | 36 | 7.44 | 56385 | 1.58 |
| nasa.xml | Astronomische Daten | 25050288 | 476646 | 56317 | 303676 | 8 | 5.08 | 2435 | 1.76 |
| psd7003.xml | Proteinsequenzen-Datenbank | 716853016 | 21305818 | 1290647 | 15955109 | 7 | 4.68 | 262529 | 1.81 |
| SwissProt.xml | Proteinsequenzen-Datenbank | 114820211 | 2977031 | 2189859 | 2013844 | 5 | 3.07 | 50000 | 2.41 |
| dblp.xml | bibliographische Daten von Publikationen | 284994162 | 6662623 | 1375832 | 6013355 | 6 | 2.39 | 649080 | 2.11 |
| customer.xml | Customers aus TPC-H Benchmark | 515660 | 13501 | 1 | 12000 | 3 | 2.41 | 1501 | 1.89 |
| ebay.xml | Ebay-Auktionsdaten | 35562 | 156 | 0 | 107 | 5 | 3.26 | 12 | 1.9 |
| lineitem.xml | Line items aus TPC-H Benchmark | 32295475 | 1022976 | 1 | 962800 | 3 | 2.45 | 60176 | 1.94 |
| mondial-3.0.xml | Geographische Datenbank aus verschiedenen Quellen | 1784825 | 22423 | 47423 | 7467 | 5 | 3.15 | 955 | 3.45 |
| orders.xml | Orders aus TPC-H Benchmark | 5378845 | 150001 | 1 | 135000 | 3 | 2.42 | 15001 | 1.9 |
| uwm.xml | Kurse einer Universitäts-Webseite | 2337522 | 66729 | 6 | 40234 | 5 | 3.37 | 2112 | 1.91 |

### 3.3.1 Speicherplatzverbrauch im *Dokumentcontainer*

Im *Dokumentcontainer* werden alle Knoten in Dokumentenordnung gespeichert. Abbildung 3.2 gibt zu jedem Dokument aus Tabelle 3.6 den durchschnittlichen Speicherplatzverbrauch pro DeweyID bei den *Distance*-Werten 2, 4, 8, 16, 32, 64, 128 und 256 an. Bei der Speicherung der DeweyIDs als Schlüssel in einem Index muss für jede DeweyID noch ein zusätzliches Byte als Längenfeld berücksichtigt werden, das in diese Auswertungen nicht mit einfließt.

**Abbildung 3.2** Durchschnittlicher Speicherplatzverbrauch pro Knoten im *Dokumentcontainer*




Bei *Distance* 2 weisen alle Dokumente die geringste Größe auf, die sich jedoch mit wachsendem Wert von *Distance* erhöht. Das hat seine Ursache in den steigenden Werten der Divisions, die durch den größeren *Distance*-Wert verursacht werden und mehr Bits zur Speicherung benötigen.

Der Speicherplatzverbrauch liegt bei *Distance* 2 zwischen 2,67 Bytes bei Dokument *ebay.xml* und 6,67 Bytes bei Dokument *treebank_e.xml*. Bei *Distance* 256 schwankt er zwischen 6,19 Bytes (*customer.xml*) und 15,94 Bytes (*treebank_e.xml*). Die *Distance*-Werte zwischen diesen Extremwerten verteilen sich gleichmäßig auf die zwischen ihnen liegenden Werte. Den stärksten Einfluss auf die Größe der DeweyIDs hat die durchschnittliche Ebene der Knoten, da Knoten einer tieferen Ebene aus entsprechend mehr Divisions bestehen, die zu einem höheren Speicherplatzverbrauch führen. Eine weitere, eher untergeordnete Rolle spielt der maximale Fanout. Durch einen hohen maximalen Fanout steigt der Speicherplatzverbrauch derjenigen DeweyIDs, die einen Knoten als Vorfahren haben, der unmittelbar an dem großen Fanout beteiligt ist, da deren große Division-Werte, die den hohen Speicherplatzbedarf verursachen, weitervererbt werden. Obwohl beispielsweise *ebay.xml* eine durchschnittliche Ebene von 3,26 besitzt und damit im Mittelfeld rangiert, liegt sein Speicherplatzverbrauch im Vergleich zu den anderen Dokumenten im unteren Bereich. Das lässt sich

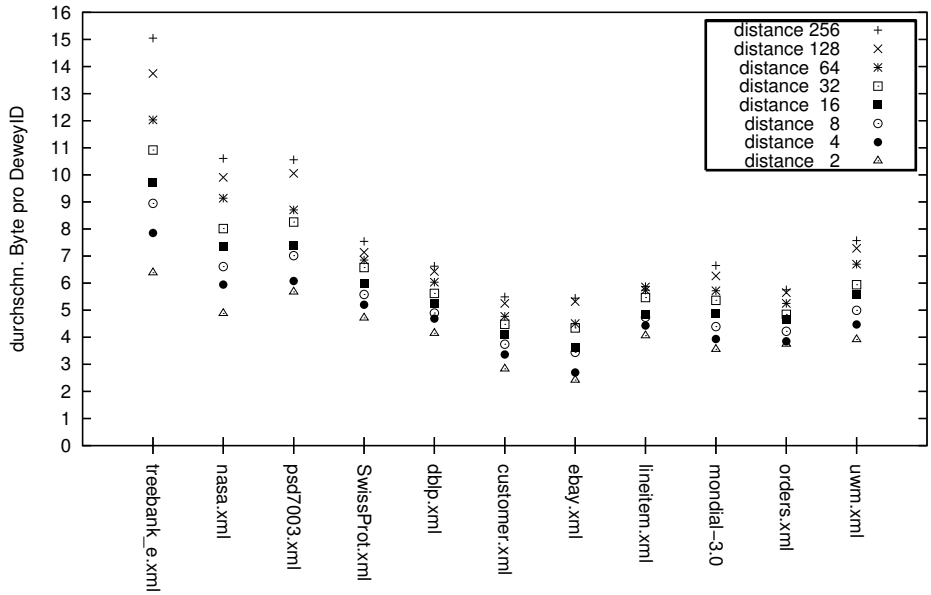
dadurch erklären, dass *ebay.xml* bei einem maximalen Fanout von 12 nur Division-Werte bis 25 (*Distance* 2) darstellen muss. Im Vergleich dazu hat *customer.xml* zwar eine um 0,85 kleinere durchschnittliche Ebene, muss also pro DeweyID weniger Divisions darstellen, besitzt dafür aber einen maximalen Fanout von 1501 und muss deshalb viele Division-Werte größer als 1000 darstellen. Das gleicht die unterschiedliche durchschnittliche Ebene beider Dokumente aus.

Auch die Dokumente *psd7003.xml* und *nasa.xml* benötigen pro DeweyID einen etwa identischen Speicherplatzverbrauch, obwohl die durchschnittliche Ebene von *nasa.xml* um 0,5 größer ist. Hier liegt der Grund ebenfalls in dem deutlich höheren maximalen Fanout von *psd7003.xml*.

### 3.3.2 Speicherplatzverbrauch im Elementindex

Im Elementindex werden alle Elementnamen gespeichert und zu jedem Elementnamen im *Knotenindex* die DeweyIDs verwaltet, die diesen Elementnamen besitzen. Abbildung 3.3 zeigt den Speicherplatzverbrauch.

**Abbildung 3.3** Durchschnittlicher Speicherplatzverbrauch pro DeweyID im Elementindex




Im Vergleich mit der vorangegangenen Grafik, die den Speicherplatzverbrauch im *Dokumentcontainer* zeigt, weisen beide Grafiken viele Ähnlichkeiten auf. Sie unterscheiden sich jedoch in der Höhe des Speicherplatzverbrauchs, der im Elementindex etwas geringer ist. Der Grund liegt in den fehlenden Text- und Attributknoten, die als Blätter des DOM-Baums immer auf einer tieferen Ebene liegen als die Elementknoten. Dadurch benötigen sie zusätzliche Divisions und erhöhen damit den durchschnittlichen Speicherplatzverbrauch im *Dokumentcontainer*.

## 3.4 Alternative Kodierungstabellen

Da die bisher verwendete Kodierungstabelle nur als Beispiel diente, werden in Abbildung 3.7 sowie in Abbildung 3.8 Alternativen vorgestellt, die in Kapitel 4.6.3 hinsichtlich ihres Speicherplatzverbrauchs untersucht werden.

**Tabelle 3.7** Kodierungstabellen mit 5 Wertebereichen

Kodierungstabelle 1

| | | |
|---|---|---|
| 0 | 7 | 1-127 |
| 10 | 14 | 128-16511 |
| 110 | 21 | 16512-2113663 |
| 1110 | 28 | 2113664-270549119 |
| 1111 | 31 | 270549120-2147483647 |

Kodierungstabelle 2

| | | |
|---|---|---|
| 0 | 3 | 1-7 |
| 10 | 6 | 8-71 |
| 110 | 13 | 72-8263 |
| 1110 | 20 | 8264-1056839 |
| 1111 | 31 | 1056840-2147483647 |

Kodierungstabelle 3

| | | |
|---|---|---|
| 0 | 3 | 1-7 |
| 10 | 6 | 8-71 |
| 110 | 13 | 72-8263 |
| 1110 | 28 | 8264-268443719 |
| 1111 | 31 | 268443720-2147483647 |

Kodierungstabelle 4

| | | |
|---|---|---|
| 0 | 3 | 1-7 |
| 10 | 6 | 8-71 |
| 110 | 21 | 72-2097223 |
| 1110 | 28 | 2097224-27053279 |
| 1111 | 31 | 27053280-2147483647 |

Kodierungstabelle 5

| | | |
|---|---|---|
| 0 | 3 | 1-7 |
| 10 | 14 | 8-16391 |
| 110 | 21 | 16392-2113543 |
| 1110 | 28 | 2113544-270548999 |
| 1111 | 31 | 270549000-2147483647 |

## 3.5 Knotenneunummerierungen

Knotenneunummerierungen werden notwendig, wenn die Länge der Byte-Repräsentation einer DeweyID eine vorher definierte Länge übersteigt, z. B. 64 Bytes. In diesem Fall wäre beispielsweise die maximale Schlüssellänge einer B*-Baum-Implementierung, in der die DeweyIDs verwaltet werden, erreicht. Die Anzahl notwendiger Neunummerierungen ist auch von der initialen Vergabe der DeweyIDs abhängig. Man kann die Anzahl der Neunummerierungen verringern, indem man die DeweyIDs so vergibt, dass es möglich ist, neue DeweyIDs einzufügen, ohne dabei gerade Divisions verwenden zu müssen, d. h. die Schrittweite *Distance* groß wählt. Somit bleibt die Länge der Byte-Repräsentation einer DeweyID geringer, da zusätzliche Divisions die Länge der DeweyID schnell erhöhen. Allerdings ist zu beachten, dass auch durch eine große *Distance* der Speicherplatzverbrauch wächst, da sich in diesem Fall die Werte der Divisions erhöhen und deshalb mehr Bits zur Speicherung benötigen. Aus diesem Grund beschäftigt sich Kapitel 3.5.1 mit der maximalen DeweyID-

**Tabelle 3.8** Kodierungstabellen mit 4 Wertebereichen

Kodierungstabelle 6

| | | |
|---|---|---|
| 0 | 7 | 1-127 |
| 10 | 14 | 128-16511 |
| 110 | 21 | 16512-2113663 |
| 111 | 31 | 2113664-2147483647 |

Kodierungstabelle 7

| | | |
|---|---|---|
| 0 | 7 | 1-127 |
| 10 | 14 | 128-16511 |
| 110 | 29 | 16512-536887423 |
| 111 | 31 | 536887424-2147483647 |

Kodierungstabelle 8

| | | |
|---|---|---|
| 0 | 7 | 1-127 |
| 10 | 22 | 128-4194431 |
| 110 | 29 | 4194432-541065343 |
| 111 | 31 | 541065344-2147483647 |

Kodierungstabelle 9

| | | |
|---|---|---|
| 0 | 3 | 1-7 |
| 10 | 6 | 8-71 |
| 110 | 13 | 72-8263 |
| 111 | 31 | 8264-2147483647 |

Kodierungstabelle 10

| | | |
|---|---|---|
| 0 | 3 | 1-7 |
| 10 | 6 | 8-71 |
| 110 | 21 | 72-2097223 |
| 111 | 31 | 2097224-2147483647 |

Kodierungstabelle 11

| | | |
|---|---|---|
| 0 | 3 | 1-7 |
| 10 | 14 | 8-16391 |
| 110 | 21 | 16392-2113543 |
| 111 | 31 | 2113544-2147483647 |

Kodierungstabelle 12

| | | |
|---|---|---|
| 0 | 3 | 1-7 |
| 10 | 14 | 8-16391 |
| 110 | 29 | 16392-536887303 |
| 111 | 31 | 536887304-2147483647 |

Kodierungstabelle 13

| | | |
|---|---|---|
| 0 | 3 | 1-7 |
| 10 | 22 | 8-4194311 |
| 110 | 29 | 4194312-541065223 |
| 111 | 31 | 541065224-2147483647 |

Länge bei den verschiedenen Dokumenten und verschiedenen *Distance*-Werten. In Unterabschnitt 3.5.2 wird anschließend das Einfügen von Teilbäumen und die dadurch ansteigende maximale Schlüssellänge beschrieben.

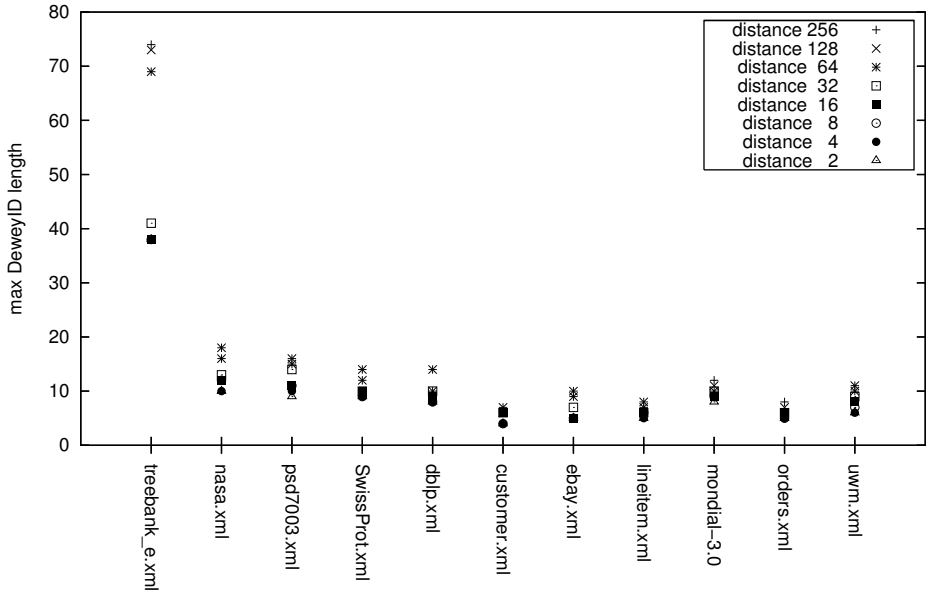
Für die folgenden Untersuchungen wird Kodierungstabelle 1 aus Abbildung 3.7 verwendet, da diese bei der Speicherung durchgängig sehr gute Ergebnisse liefert.

### 3.5.1 Maximale DeweyID-Länge

Die maximale DeweyID-Länge ist neben der durchschnittlichen Länge ein weiteres wichtiges Kriterium bei Knotennummerierungen, auch wenn bei den meisten DeweyIDs durch die Präfix-Komprimierung nur wenige Bytes gespeichert werden müssen, da die meisten Bytes vom vorherigen Schlüssel übernommen werden können. Zum einen, da eine einzelne große DeweyID eine Neunummerierung verursachen kann und zum anderen, weil der erste Schlüssel einer Seite immer vollständig gespeichert werden muss und somit zusätzlichen Speicherplatz verursacht. In Abbildung 3.4 sind die maximalen DeweyID-Längen grafisch dargestellt.

Wenn man *treebank_e.xml* außer Betracht lässt, reichen zur Speicherung der größten De-

**Abbildung 3.4** Maximale DeweyID-Länge




weyID 18 Bytes. *treebank_e.xml* hat die mit Abstand größte maximale Ebene (36), was erklärt, warum hier sehr große DeweyIDs entstehen. Wenn jede dieser Ebenen ein Byte benötigt, sind bereits 36 Bytes für die vollständige Speicherung der DeweyID erforderlich (bei *Distance* 2 sind es 38 Bytes). Bis zu *Distance* 256 steigt die maximale Größe bei *treebank_e.xml* auf 74 Bytes. Bei den anderen Dokumenten erreicht die größte DeweyID bei *Distance* 2 zwischen 4 und 10 Bytes und steigt bei *Distance* 256 auf 7 bis 18 Bytes an. Außer bei sehr tiefen Dokumenten spielt die maximale DeweyID-Länge damit keine große Rolle. Bei tiefen Dokumenten wie *treebank_e.xml* muss der *Distance*-Wert sorgfältig gewählt werden, da die maximale DeweyID-Größe sprunghaft ansteigt. In Abbildung 4.29 aus Kapitel 4.6, das die maximale DeweyID-Länge verschiedener Kodierungsmöglichkeiten nochmals genau untersucht, wird die maximale DeweyID-Länge von Kodierung K1 mit Kodierung K3 bei *treebank_e.xml* verglichen, wo die Sprünge deutlich sichtbar sind. Die maximale DeweyID-Länge der anderen Dokumente ist zu diesen beiden Kodierungen im Anhang B abgebildet. Die Folge des sprunghaften Anstiegs ist, dass bei einem zu hohen *Distance*-Wert weniger Einfügungen möglich sind als bei einem geringeren *Distance*-Wert. Bei Verwendung der Kodierungstabelle K1 ist 30 ein geeigneter *Distance*-Wert, da dadurch die ersten vier Kinder eines Knotens nur ein Byte benötigen. Sie bekommen die DeweyID des Vaterknotens mit den zusätzlichen Divisions 31, 61, 91 und 121. Bei *Distance*-Wert 32, bei dem die ersten vier Knoten die Divisions 33, 65, 97 und 129 angehängt bekommen, werden für den vierten Knoten bereits zwei Bytes verwendet, da er in den zweiten Wertebereich fällt.

### 3.5.2 Einfügungen von Teilbäumen

Die Anzahl der Bits, die zur Speicherung von Divisions benötigt werden, kann man aus der Kodierungstabelle errechnen, indem man die Anzahl der Bits für $L_i$ und die Anzahl der Bits

für $O_i$ des entsprechenden Wertebereichs addiert. Eine Division mit einem Wert zwischen 1 und 127 braucht beispielsweise acht Bits, eine Division mit einem Wert zwischen 128 und 16511 entsprechend 16 Bits.

Zur Untersuchung des Anstiegs der maximalen Schlüssellänge wird ein Dokument angenommen, das bereits gespeichert ist und bei dem weitere Teilbäume auf Ebene 1 eingefügt werden. Um dabei den *worst-case* zu betrachten, sollen die eingefügten Teilbäume die gleiche Struktur annehmen wie der Teilbaum des Dokuments mit dem Knoten, der die größte DeweyID des gesamten Dokuments besitzt. Durch diese Struktur entsteht in dem neuen Teilbaum ein Knoten, der gleich groß oder größer ist als der bis dahin größte Knoten. Es entsteht genau dann ein größerer Knoten, wenn für den Teilbaum eine gerade Division verwendet werden muss. So wächst also der Speicherverbrauch der Ebene 1 durch die Verwendung zusätzlicher gerader Divisions. Wenn z. B. die größte DeweyID 1.1343.5.7.3.3 lautet, könnte als neue größte DeweyID die 1.1344.5.5.7.3.3 entstehen. Da stets der gleiche Teilbaum eingefügt wird und sich dadurch die Divisions von Ebene 2 an nicht verändern, kann zur Vereinfachung der Speicherplatz für die Divisions ab Ebene 2 pauschal berechnet werden.

Als Beispiel dient das Dokument *treebank_e.xml,* das von den untersuchten Dokumenten mit der maximalen Ebene 36 die größte Herausforderung darstellt, und auch die größte maximale DeweyID-Länge besitzt.

Das Dokument wird mit *Distance* 32 gespeichert, bei der die maximale DeweyID-Länge 41 Bytes beträgt. Diese DeweyID hat die Form 1.x.y, wobei x auf der Ebene liegt, in der die Teilbäume eingefügt werden und y den Teilbaum repräsentiert, der darunter liegt. Zwischen den Knoten 1.x und 1.x+32 können mindestens vier Teilbäume eingefügt werden, ohne eine zusätzliche gerade Division verwenden zu müssen, wie folgende Worst-case-Einfügung zeigt: 1.x+16.y, 1.x+8.y, 1.x+4.y und 1.x+2.y. Da zwischen 1.x und 1.x+2 keine gerade Division mehr passt, muss nach Regeln 3.1.2.3 eine gerade Division mit einer weiteren Division (*Distance*+1*)* verwendet werden. Der neue Teilbaum ist also *1*.x+1.33.y. Der Speicherplatzverbrauch einer DeweyID hat sich jetzt um ein Byte erhöht, da die zusätzliche Division 33 ein Byte benötigt. Die maximale DeweyID-Länge ist also auf 42 Bytes gestiegen. Jetzt können mindestens weitere vier Knoten eingefügt werden, ohne eine gerade Division verwenden zu müssen: 1.x+1.17.y, 1.x+1.9.y, 1.x+1.5.y und 1.x+1.3.y. Danach wird wieder eine gerade Division benötigt: 1.x+1.2.33.y. Es können also für ein zusätzliches Byte immer fünf neue Teilbäume eingefügt werden. Bei einer maximalen Schlüssellänge von 128 ergeben sich in diesem Beispiel insgesamt 439 Teilbäume. Die Anzahl der einfügbaren Teilbäume nach diesem Verfahren beschreibt folgende Formel:

$$v \quad = \quad (maxDL - maxKey + 1) * log_2(distance) - 1$$

Dabei ist:

**v:** einfügbare Teilbäume

**maxDL:** maximale DeweyID-Länge

**maxKey:** maximale Schlüssellänge im bisherigen Dokument

# Kapitel 4 Speicherungsstrukturen für XML-Dokumente

Der XML Transaction Coordinator ist durch das taDOM-Datenmodell, das in Kapitel 2.2.1 vorgestellt wurde, darauf ausgelegt, XML-Dokumente auf Knotenebene zu verarbeiten. Die Knoten des Dokuments werden zusammen mit ihrer DeweyID in Dokumentenordnung in einer Liste gespeichert. Das Speichern des Dokuments in Dokumentenordnung hat mehrere Vorteile. Zum Beispiel liegt das Dokument in einer Textdatei bereits in dieser Ordnung vor und muss beim *bulkloading* die bestehende Reihenfolge nur beibehalten. Außerdem wird auf diese Weise der Teilbaum eines Kontextknotens unmittelbar hinter dem Kontextknoten gespeichert, so dass die Abfrage eines Teilbaums effizient durchgeführt werden kann. Davon profitiert z. B. der SAX-Parser, der das Dokument immer sequentiell verarbeitet. Um einen Knoten, dessen DeweyID bekannt ist, schnell auffinden zu können, sind die Seiten der Liste durch einen B-Baum indexiert. Das bedeutet, dass im B-Baum die erste DeweyID jeder Seite verwaltet wird. Wird ein Knoten anhand seiner DeweyID gesucht, so kann mit Hilfe des B-Baums schnell die Seite gefunden werden, in der er sich befindet. Das Speichern der Daten in verketteten Seiten mittels einer Indexierung über einen B-Baum entspricht im Prinzip einem B*-Baum, der im Indexmanager implementiert wird. Das Speichern der Knoten übernimmt der Satzmanager, der die physischen Datensätze erstellt (siehe Abbildung 2.4) und im Indexmanager ablegt.

Der Indexmanager ist Teil der Zugriffsdienste (siehe Abbildung 2.1), die auf dem Puffermanager aufbauen. Somit besteht für die Zugriffsdienste die Möglichkeit, transaktionskonsistent neue Seiten vom Puffermanager anzufordern, gespeicherte Seiten zu laden, Seiten zu verändern oder zu löschen. Der Indexmanager hat die Aufgabe, einen Index zu erstellen, Schlüssel mit dem dazugehörigen Wert darin zu speichern, den Wert eines Schlüssels zu suchen und bei Bedarf wieder zu löschen. Auch muss er in der Lage sein, den Wert eines Schlüssels zu ändern. Schlüssel und Werte können dabei vom Datentyp DeweyID, String, Integer oder Byte sein. Der Indexmanager muss variable Schlüssel- und Wertlängen unterstützen, da Schlüssel und Werte sehr stark in der Größe variieren. Es existieren drei verschiedene Indextypen (Liste, B-Baum und B*-Baum), aus denen beim Erstellen des Indexes über Parameter ausgewählt werden muss. Der B*-Baum ist die oben beschriebene Indexierung einer Liste durch einen B-Baum.

Im nächsten Abschnitt werden zunächst die wichtigsten öffentlichen Methoden des Indexmanagers erklärt, die dieser implementieren muss. Abschnitt 4.2 gibt einen Überblick über

die Operationen des B-Baums und die Realisierung des B*-Baums aus Liste und B-Baum. Die Unterschiede zwischen der bereits im XTC-Server implementierten Version, hier bezeichnet als Indexmanager 1, und dem neu implementierten Indexmanager, hier genannt Indexmanager 2, werden in Abschnitt 4.3 beschrieben. Die Implementierungsdetails in Abschnitt 4.4 geben Auskunft über Seiten- und Datenformate. Der darauf folgende Abschnitt beschäftigt sich mit der Transaktionsisolation, d. h., welche Arten von Sperren auf Seitenebene existieren und wie diese eingesetzt werden, um Transaktionsisolation zu gewährleisten. In diesem Zusammenhang wird auch erklärt, wie mehrere Transaktionen gleichzeitig auf eine Indexstruktur zugreifen können. Abschnitt 4.6 beschäftigt sich ausführlich mit dem Speicherplatzverbrauch der verwendeten Implementierung. Abschließend werden Speicherplatzverbrauch und Leistung der beiden Indexmanager gegenübergestellt.

## 4.1 Methoden des Indexmanagers

Der Indexmanager bietet folgende öffentliche Methoden zur Verwaltung von Indizes an. Schlüssel und Werte werden grundsätzlich in ihrer Byte-Repräsentation übergeben und zurückgeliefert, so dass die aufrufende Methode zur Kodierung und Dekodierung verantwortlich ist. Unter *transaction* referenziert die aufrufende Methode die Transaktion, unter der die Operation ausgeführt wird.

- `public int createIndex (XTCtransaction transaction, int containerNo, int idxType, int keyType, int valueType)`
  Erstellt einen neuen Index in dem übergebenen Container. *idxType* bestimmt den Indextyp und kann vom Typ *INDEXTYPE_BTREE, INDEXTYPE_LIST* oder *INDEXTYPE_BSTREE* sein. *keyType* und *valueType* bestimmen den Schlüsseltyp bzw. Werttyp und können vom Typ *VALUETYPE_INT, VALUETYPE_STRING, VALUETYPE_BYTES* oder *VALUETYPE_DEWEYID* sein. Die Methode liefert eine Indexnummer zurück, mit welcher der Index eindeutig identifiziert werden kann.

- `public byte[] getFromIndex (XTCtransaction transaction, int idxNo, byte[] searchKey)`
  Liefert einen zuvor eingefügten Wert des übergebenen Schlüssels *searchKey* aus dem Index *idxNo* zurück. Wenn der Schlüssel nicht im Index existiert, wird *Null* zurückgeliefert.

- `public void insertIntoIndex (XTCtransaction transaction, int idxNo, byte[] newKey, byte[] newValue)`
  Fügt den Schlüssel *newKey* mit dem Wert *newValue* in den Index *idxNo* ein.

- `public void updateIndex (XTCtransaction transaction, int idxNo, byte[] key, byte[] oldValue, byte[] newValue)`
  Ändert im Index *idxNo* den Wert *oldValue* des Schlüssels *key* in *newValue*.

- `public void deleteFromIndex (XTCtransaction transaction, int idxNo, byte[] key, byte[] oldValue)`
  Löscht aus dem Index *idxNo* den Schlüssel *key* mit dem Wert *oldValue*.

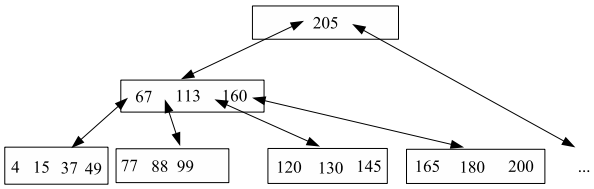
- `public void deleteIndex (XTCtransaction transaction, int idxNo)`
  Löscht den Index *idxNo*.

## 4.2 B*-Bäume

Zum Speichern großer Datenmengen in Datenbanksystemen haben sich B-Bäume bzw. Variationen (B+-Bäume und B*-Bäume) bewährt [5]. Auch im XTC-Server wird diese Technik verwendet, um Daten auf dem Permanentspeicher abzulegen. Ziel ist es, die Daten so zu strukturieren, dass gesuchte Datensätze möglichst schnell aufgefunden und eingefügt werden können und gleichzeitig eine möglichst hohe Speicherplatzausnutzung erreicht wird.

Ein Beispiel eines B-Baums ist in Abbildung 4.1 dargestellt.

**Abbildung 4.1** B-Baum




Der B-Baum besteht aus Knoten (Seiten), die Schlüssel/Wert-Paare enthalten. Die Seiten speichern die Datensätze, die in der Abbildung durch einfache Zahlen dargestellt werden, in Sortierreihenfolge der Schlüssel. Dieser ist balanciert, d. h., die Blattknoten liegen alle auf gleicher Ebene. Außerdem muss eine Seite mindestens halb gefüllt sein. Eine Ausnahme bildet der Wurzelknoten, der auch weniger als halb gefüllt sein darf. Rechts und links jedes Schlüssels sind Verweise auf Seiten gespeichert, die kleinere (links) oder größere (rechts) Schlüssel beinhalten. Diese werden durch die Pfeile in der Darstellung angezeigt. Außerdem verweist eine Seite ebenfalls auf ihre Elternseite.

Die Suche eines Schlüssels beginnt in der Wurzelseite. In ihr wird der erste Schlüssel gesucht, der größer oder gleich dem zu suchenden Schlüssel ist. Entweder findet man den Schlüssel in der Seite, oder man setzt die Suche mit dem gleichen Verfahren in der Seite fort, die durch den Verweis links des Schlüssels adressiert wird. Eine erfolglose Suche endet in einer Blattseite. Eine erfolgreiche Suche kann bereits in einer inneren Seite abgeschlossen sein, endet aber meist auch in einer Blattseite, da dort die meisten Datensätze liegen.

Damit der B-Baum auch nach dem Einfügen und Löschen von Datensätzen balanciert ist, müssen weitere Maßnahmen ergriffen werden. Das Einfügen eines Schlüssel/Wert-Paares geschieht in zwei Phasen. Zuerst muss die Position gesucht werden, an der die Einfügung des neuen Datensatzes erfolgen soll. Dazu wird der Suchalgorithmus auf den neuen Schlüssel angewendet, dessen erfolglose Suche in dem Blatt endet, in dem der neue Schlüssel eingefügt werden muss. Danach kann der Schlüssel an dieser Position eingefügt werden, sofern die Seite über genügend freien Speicherplatz verfügt (in Abbildung 4.2 wurde der Schlüssel 100 eingefügt). Ansonsten muss die Seite geteilt werden. Das Teilen der Seite wirkt sich in

Bottom-up-Richtung aus, also von den Blättern zur Wurzel. Dazu wird der erste Datensatz der zweiten Hälfte (*Median*) gesucht und alle Datensätze hinter diesem *Median* in eine neue Seite gespeichert. Der *Median* wird in der Elternseite gespeichert, zusammen mit einem Verweis auf die neue Seite, welche die Datensätze der zweiten Hälfte enthält. Falls die Seite, in die der *Median* gespeichert werden muss, zu klein ist, muss auch diese nach dem gleichen Verfahren geteilt werden. Dieser Vorgang kann sich bis zur Wurzel fortsetzen. Muss die Wurzel geteilt werden, bildet der *Median* die neue Wurzel. Als Beispiel wird in den B-Baum aus Abbildung 4.2 der Schlüssel 101 eingefügt. Das Ergebnis zeigt Abbildung 4.3. Die Seite mit den Schlüsseln „77 88 99 100“ musste geteilt werden, da der neue Datensatz nicht in die Seite passt. Der *Median* 99 wandert in die Elternseite und trennt die beiden Hälften der Seite.

**Abbildung 4.2** B-Baum nach Einfügen des Schlüssels 100

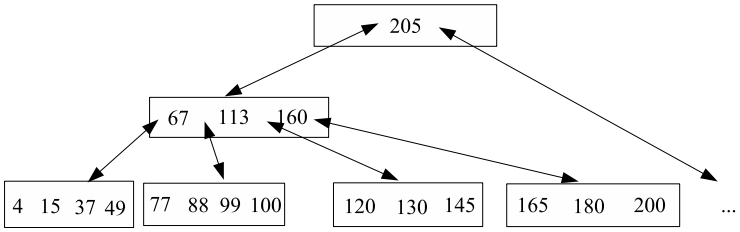
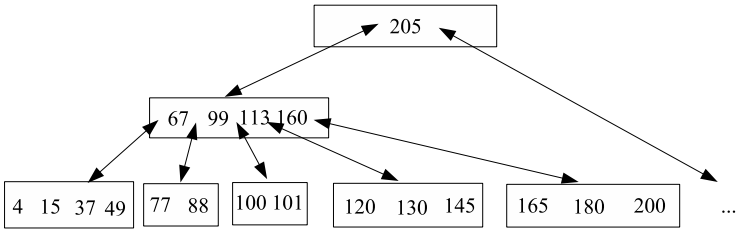


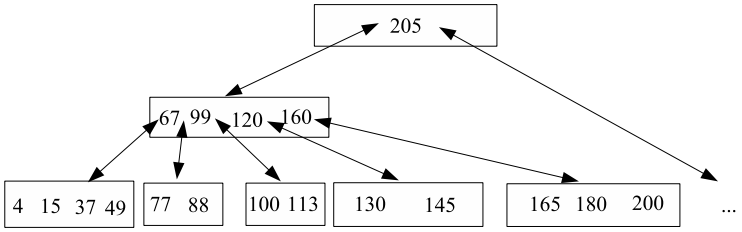

**Abbildung 4.3** B-Baum nach Einfügen des Schlüssels 101




Beim Löschen von Datensätzen muss zwischen dem Löschen in einer Blattseite und dem Löschen in einer inneren Seite unterschieden werden. Falls sich der zu löschende Datensatz in einer Blattseite befindet, kann er direkt gelöscht werden. Ist nach dem Löschvorgang die Seite weniger als halb gefüllt, findet entweder eine Verschmelzung von Seiten oder eine Verschiebung von Datensätzen statt, um sicherzustellen, dass wieder alle Seiten mindestens zur Hälfte gefüllt sind. Zuerst wird die Balancierungsseite festgelegt. Standardmäßig ist das die rechte Nachbarseite. Wenn rechts keine Seite existiert, wird die linke Seite verwendet. Sofern die beiden Seiten, zusammen mit dem Schlüssel der Elternseite, welcher die Wertebereiche der Schlüssel der beiden Seiten trennt, in eine Seite passen, findet eine Verschmelzung statt. In diesem Fall wird der gewählte Schlüssel der Elternseite gemeinsam mit der rechten Seite in die linke Seite verschoben. Der Verweis rechts neben dem Schlüssel der Elternseite wird gelöscht. Kann keine Verschmelzung stattfinden, wird ein Schlüssel der Balancierungsseite in die unterfüllte Seite verschoben. Ist die rechte Seite die Balancierungsseite, wird der Schlüssel der Elternseite, der die Wertebereiche trennt, hinter den letzten Schlüssel

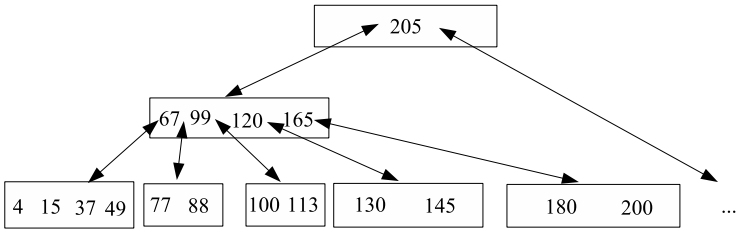
der unterfüllten Seite verschoben und der erste Schlüssel der Balancierungsseite an dessen Position verschoben. Ist dagegen die linke Seite die Balancierungsseite, wird der Schlüssel der Elternseite, der die Wertebereiche trennt, vor den ersten Schlüssel der unterfüllten Seite verschoben und der letzte Schlüssel der Balancierungsseite an dessen Position verschoben. In Abbildung 4.4 wurde der Schlüssel 101 gelöscht. 113 wandert an dessen Position und der Datensatz mit Schlüssel 120 an die frei gewordene Position in die Elternseite.

**Abbildung 4.4** B-Baum nach dem Löschen von 101




Wenn sich der zu löschende Schlüssel nicht in einer Blattseite, sondern in einer inneren Seite befindet, sind weitere Maßnahmen erforderlich, da dieser Schlüssel die Verweise zu den Unterbäumen trennt. Deshalb wird nach dem Löschen des Datensatzes der nächst größere Schlüssel, der sich immer im rechten Teilbaum im Blattknoten ganz links befindet, an die Stelle des gelöschten Schlüssels verschoben. Ein Unterlauf der Seite wird ebenfalls mit Verschmelzung von Seiten oder Verschiebung von Datensätzen behandelt. Abbildung 4.5 zeigt den B-Baum nach dem Löschen des Datensatzes mit dem Schlüssel 160. Der Datensatz mit dem Schlüssel 165 wandert dabei hoch in den Elternknoten.

**Abbildung 4.5** B-Baum nach dem Löschen von 160

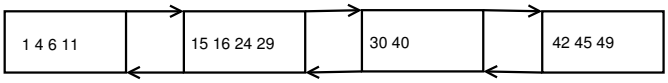



Ein B*-Baum ist eine Variante des B-Baums, bei der in inneren Seiten keine Datensätze gespeichert werden, sondern diese nur in den Blattseiten liegen. Die Schlüssel der inneren Seiten dienen lediglich als Wegweiser zu den Blattknoten. Da in die inneren Seiten des B*-Baums mehr Datensätze passen, hat dieser gegenüber dem B-Baum den Vorteil, dass eine höhere Verzweigung erreicht wird und sich damit die Höhe des Baums verringert. Als positiver Effekt enstehen weniger Seitenzugriffe.

Der B*-Baum ist in der Implementierung nicht exakt so aufgebaut wie in der Literatur beschrieben, verhält sich aber sehr ähnlich. Die Datenseiten des B*-Baums werden durch eine doppelt verkettete Liste wie in Abbildung 4.6 implementiert.

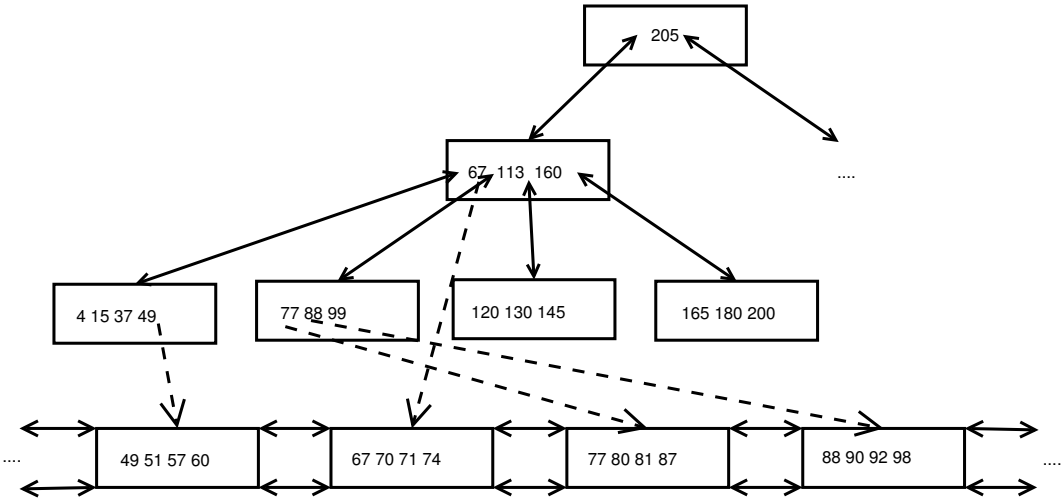
Diese Liste wird von einem B-Baum indexiert. Dazu wird zusätzlich zu der Liste, die die ei-

**Abbildung 4.6** Doppelt verkettete Liste




gentlichen Datensätze speichert, ein B-Baum gepflegt. In ihm wird jeweils der erste Schlüssel jeder Seite der Liste zusammen mit einem Verweis auf die Seite, in der er enthalten ist, gespeichert. Abbildung 4.7 zeigt das Verfahren ausschnittsweise, wobei die gestrichelten Pfeile die Verweise symbolisieren.

**Abbildung 4.7** B*-Baum




Bei der Suche eines Schlüssels wird erst der größte Schlüssel im B-Baum gesucht, der kleiner oder gleich dem zu suchenden Schlüssel ist. Falls der gesuchte Schlüssel in der Liste existiert, muss er in der Seite enthalten sein, auf die der Schlüssel des B-Baums zeigt. Wird beispielsweise der Schlüssel 71 gesucht, so wird als größter Schlüssel im B-Baum, der kleiner ist als 71, die 67 gefunden, die auf die Seite verweist, in der die 71 liegt. Wäre die Liste nicht indexiert, müsste eine Suche immer bei der ersten Seite beginnen. Durch eine Indexierung aber wird die Anzahl der Seitenzugriffe im Vergleich zu denen einer Liste erheblich reduziert, da immer nur wenige Zugriffe im B-Baum nötig sind, d. h. meist nicht mehr als drei bis vier, und anschließend direkt ein Zugriff auf die Seite der Liste erfolgen kann.

Wird der erste Schlüssel gelöscht, so muss dieser ebenfalls aus dem B-Baum entfernt und an seine Stelle der nachfolgende Schlüssel eingetragen werden.

Vor dem Einfügen eines Schlüssels wird, wiederum mit Hilfe des B-Baums, die Seite in der Liste gesucht, in die der Schlüssel eingesetzt werden muss. Falls für den Schlüssel nicht genügend Platz vorhanden ist, wird die Seite geteilt. Dann muss der erste Schlüssel der zweiten Hälfte in den B-Baum eingefügt werden.

## 4.3 Vergleich zum vorhandenen Ansatz

Indexmanager 1 bezeichnet die bereits im XTC-Server implementierte Version, die als Grundlage für eine neue Version, Indexmanager 2, dient. Der Indexmanager 2 erfuhr dabei mehrere Verbesserungen. Zum einen wurde das Datensatzformat komplett überarbeitet. Im Indexmanager 1 wurden aus einem Schlüssel/Wert-Paar zwei Datensätze erstellt. Es musste zuerst ein Datensatz für den Schlüssel und dahinter ein zweiter Datensatz mit dem zugehörigen Wert gespeichert werden. Diese Trennung wurde im Indexmanager 2 aufgegeben. Die Datensatzformate stellt Abschnitt 4.4.2 vor.

Zum anderen implementiert der Indexmanager 2 eine Präfixschlüssel-Komprimierung, durch die bei Speicherung der DeweyIDs sehr viel Speicherplatz eingespart werden kann. Eine weitere grundlegende Änderung betrifft die Sperrverwaltung beim Traversieren des B-Baums. Während im Indexmanager 1 bei Modifikationen alle Seiten gesperrt blieben, bis die Indexoperation beendet ist, werden im Indexmanager 2 die Seiten schon entsperrt, sobald sichergestellt ist, dass die Seitensperre nicht mehr benötigt wird. Das Vorgehen bei dieser Strategie zeigt Abschnitt 4.5.1.

## 4.4 Implementierung

### 4.4.1 Seitenformate

In diesem Abschnitt werden die für diese Arbeit relevanten Seitenformate abgebildet und erläutert. Die ersten elf Bytes jeder Seite besitzen das Standard-Seitenformat, d. h., sie sind in jeder Seite identisch. Im Anschluss folgt für jeden Seitentyp ein individuelles Format.

#### Standard-Seitenformat

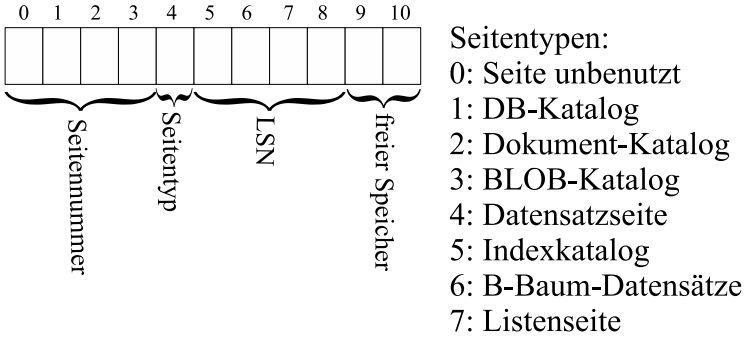
Abbildung 4.8 zeigt das Seitenformat, das jede Seite im XTC-Server verwendet. Die ersten vier Bytes speichern die Seitennummer. Im darauf folgenden Byte wird der Seitentyp definiert. Die Log Sequence Number, die fünf Bytes in Anspruch nimmt, dient zur Transaktionsverwaltung. Auf sie wird aber in dieser Arbeit nicht näher eingegangen. In den folgenden zwei Bytes wird der freie Speicherplatz der Seite verwaltet.

Nach diesen elf Bytes folgen abhängig vom Seitentyp weitere Bytes. Die wichtigsten Seitentypen mit ihrer jeweiligen Byte-Verteilung werden im Folgenden dargestellt.

#### Seitenformat im Indexkatalog

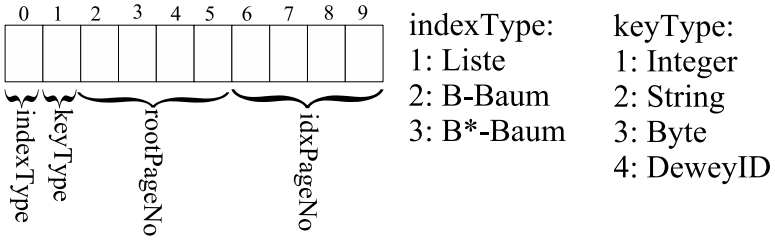
Abbildung 4.9 veranschaulicht, wie die Seite des Indexkatalogs aufgebaut ist. In *indexType* wird gespeichert, um welchen Indextyp es sich handelt (Liste, B-Baum oder B*-Baum). Im folgenden Byte ist der Schlüsseltyp definiert (Integer, String, Byte oder DeweyID). Die nächsten vier Bytes beschreiben die Wurzelseite des Indexes. Bei einem B-Baum ist das der Wurzelknoten, bei Liste und B*-Baum hingegen die erste Seite der Liste, d. h. im B*-Baum

**Abbildung 4.8** generelles Seitenformat




die erste Seite der Datenseiten. Bei B*-Bäumen wird in den nachfolgenden vier Bytes in *idxPageNo* die Indexnummer des B-Baums gespeichert, der die Liste indexiert. Bei B-Baum und nicht indexierter Liste ist die *idxPageNo* gleich 0.

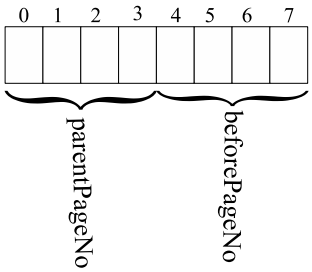
**Abbildung 4.9** Seitenformat vom Indexkatalog




### Seitenformat im B-Baum

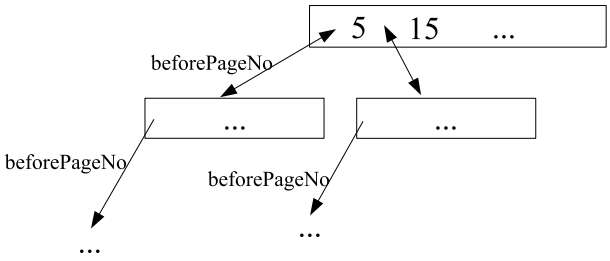
Abbildung 4.10 zeigt, wie die Datenseiten eines B-Baums aussehen.

**Abbildung 4.10** Seitenformat von Datenseiten eines B-Baums




In *parentPageNo* wird die Seitennummer der Elternseite gespeichert und in *beforePageNo* die Seitennummer, welche die Datensätze enthält, die kleiner als der kleinste Schlüssel der Seite sind (siehe Abbildung 4.11). Die restlichen Verweise zu Sohnseiten speichert das Datenformat. Dabei existiert zu jedem Datensatz eine Sohnseite (siehe auch Datenformate im B-Baum in Abschnitt 4.4.2).

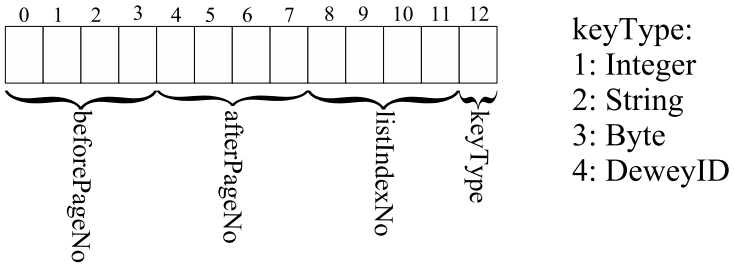
**Abbildung 4.11** beforePageNo




### Seitenformat in der Liste

In der Seite einer Liste werden, wie in Abbildung 4.12 gezeigt, vier Bytes verwendet, um die vorherige Seite zu speichern und weitere vier Bytes, um die nachfolgende Seite zu speichern. In *listIndexNo* wird die Indexnummer des B-Baums gespeichert, sofern die Liste durch einen B-Baum indexiert wurde. Der *keyType* speichert, von welchem Typ die Schlüssel sind (Integer, String, Bytes oder DeweyID).

**Abbildung 4.12** Seitenformat von Datenseiten einer Liste



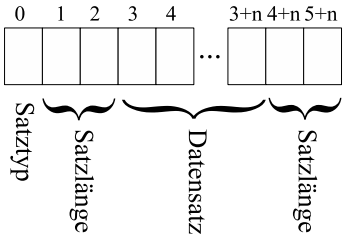

## 4.4.2 Record-Formate

### Datenformat in der Liste

Im Indexmanager 1 müssen für ein Schüssel/Wert-Paar zwei Datensätze gespeichert werden, nämlich ein Datensatz für den Schlüssel und ein zweiter Datensatz für den Wert. Bei einem Datensatz wird an der ersten Byte-Position angegeben, ob dieser Datensatz ein Schlüssel oder ein Wert ist. Zur Speicherung eines Schlüssel/Wert-Paares wird also erst ein Datensatz mit dem Schlüssel gespeichert und nachfolgend ein Datensatz mit dem dazugehörigen Wert. Danach wird in den nächsten zwei Bytes die Länge des Datensatzes, d. h. entweder des Schlüssels oder des Werts, gespeichert, so dass die maximale Länge eines Schlüssels bzw. Werts 64 Kilobytes beträgt. Bei Byte-Position drei beginnt nun die Speicherung des Datensatzes. Am Ende des Datensatzes wird nochmals die Satzlänge gespeichert. Dadurch ist es möglich, die Datensätze einer Seite rückwärts zu lesen. Wenn die Position eines Datensatzes bekannt ist, kann der vorherige Datensatz gelesen werden, indem die Satzlänge, d. h. die

zwei Bytes unmittelbar vor dem gerade gelesenen Datensatz, erfasst und die Byte-Position des Datensatzbeginns errechnet wird. Das Datensatzformat ist in Abbildung 4.13 abgebildet.

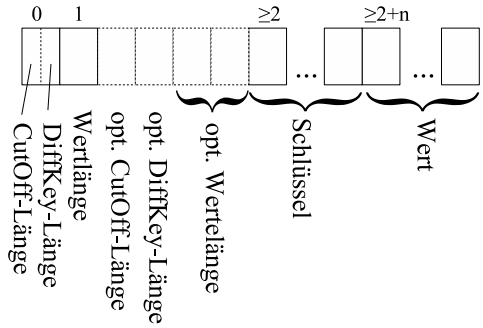
**Abbildung 4.13** Datensatzformat der Liste im Indexmanager 1




Die Benutzung von eigenen Datensätzen zur Speicherung von Schlüssel und Wert bringt dann Vorteile, wenn zu einem Schlüssel viele Werte gespeichert werden müssen. Falls jedoch zu jedem Schlüssel nur ein einziger Datensatz gespeichert wird, ist der Speicherplatzverbrauch aufgrund der expliziten Angabe des Satztyps höher. Da im XTC-Server zu jedem Schlüssel nur ein Wert existiert, ist es vorteilhaft, Schlüssel und Wert in einem Datensatz zu speichern.

Wie in Kapitel 4.6 noch zu sehen sein wird, lässt sich die Präfix-Komprimierung in *Dokumentcontainer* und *Elementindex* sehr effektiv anwenden. Diese Erkenntnis fließt auch in das Datensatzformat des Indexmanagers 2 ein, indem im Schlüssel nur noch der Unterschied zum vorherigen Datensatz gespeichert wird. Da zur Dekodierung eines Schlüssels wegen der Präfix-Komprimierung der vorherige Schlüssel bekannt sein muss, macht die Speicherung der Satzlänge am Ende eines Datensatzes keinen Sinn. Aus diesem Grund wird das Datensatzformat aus Abbildung 4.14 gewählt.

**Abbildung 4.14** Datensatzformat der Liste im Indexmanager 2




Ein Datensatz besteht aus den folgenden Komponenten:

- CutOff-Länge
- DiffKey-Länge
- Wertlänge

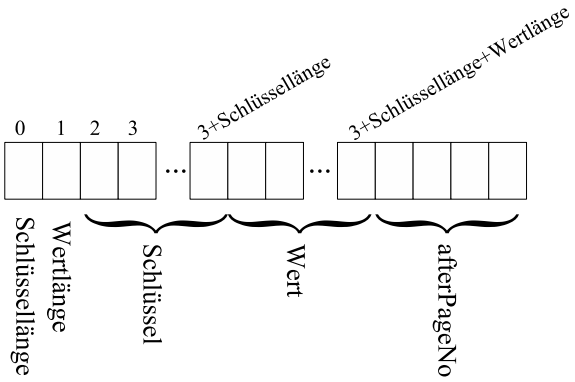
- Schlüssel
- Wert

Die CutOff-Länge beschreibt, wie viele Bytes vom vorangehenden Schlüssel abgeschnitten werden müssen. Die DiffKey-Länge definiert, wie viele Bytes zu dem abgeschnittenen Schlüssel noch hinzukommen. Diese zusätzlichen Bytes werden im Schlüssel gespeichert. Die Wertelänge gibt die Länge des Werts an.

### Datenformat im B-Baum

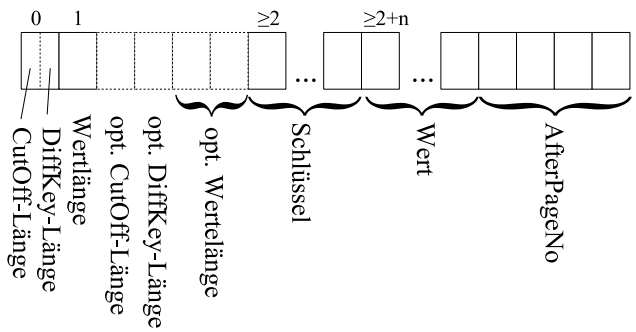
Das Datensatzformat im B-Baum in Indexmanager 1 zeigt Abbildung 4.15. Es speichert die Schlüssellänge, die Wertelänge, den Schlüssel, den Wert und die so genannte *AfterPage-Number*. In der *AfterPageNumber* wird der Verweis zur Sohnseite gespeichert, die größere Schlüssel enthält, als der aktuelle Datensatz beinhaltet, und kleinere Schlüssel als der nachfolgende Datensatz. Für den Verweis vor dem ersten Datensatz existiert in jeder B-Baum-Seite noch ein reservierter Speicherbereich.

**Abbildung 4.15** Datensatzformat im B-Baum von Indexmanager 1




Das Datensatzformat im B-Baum im Indexmanager 2 ist fast identisch mit dem Datensatzformat in der Liste, mit dem einzigen Unterschied, dass am Ende noch die *AfterPageNumber* gespeichert wird (siehe Abbildung 4.16).

**Abbildung 4.16** Datensatzformat im B-Baum von Indexmanager 2



## 4.5 Transaktionsisolation

Beim Suchen, Einfügen und Löschen eines Schlüssels im B-Baum, B*-Baum oder in der Liste müssen die besuchten Seiten zur Synchronisation gesperrt werden. Dazu existieren drei verschiedene Sperren: *read*, *update* und *exclusive*. Die *read*-Sperre wird bei Lesetransaktionen eingesetzt. Mehrere Lesetransaktionen können gleichzeitig auf gemeinsame Seiten zugreifen. Die *update*-Sperre kommt bei geplanten oder möglichen Änderungsoperationen zum Einsatz. Wenn tatsächlich eine Änderung stattfindet, wird die *update*-Sperre zu einer *exclusive*-Sperre erhöht. Tabelle 4.1 zeigt die Sperrkompatibilitätsmatrix. In der Kopfzeile steht die bereits existierende Sperre einer Transaktion und in der Kopfspalte die angeforderte Sperre einer weiteren Transaktion. Falls eine *read*-Sperre existiert, sind weitere *read*-Sperren möglich. Außerdem kann bei einer *read*-Sperre eine andere Transaktion eine *update*-Sperre erhalten. Wenn eine Transaktion aber eine *update*-Sperre hält, können keine weiteren *read*-Sperren mehr vergeben werden. Soll die *update*-Sperre in eine *exclusive*-Sperre umgewandelt werden, muss gewartet werden, bis alle *read*-Sperren anderer Transaktionen freigegeben wurden.

**Tabelle 4.1** Sperrkompatibilitätsmatrix der Seitensperren

| | | R | U | X |
|---|---|---|---|---|
| R | + | + | - | - |
| U | + | + | - | - |
| X | + | - | - | - |

### 4.5.1 Vorzeitiges Entsperren

Wenn ein neuer Schlüssel eingefügt wird, kann es zu Split-Operationen kommen, wie in Kapitel 4 beschrieben. Eine Split-Operation kann sich bis zur Wurzel fortsetzen. Falls allerdings eine Datenseite genug freien Speicherplatz besitzt, um einen Datensatz zu speichern, der die maximale Schlüssellänge und Wertelänge hat, endet spätestens bei diesem Knoten die Split-Operation. Um einen neuen Datensatz einzufügen, muss zur Suche der Position erst die Seite des Wurzelknotens im Katalog des Indexes gelesen werden. Die Katalogseite muss bei dem Vorgang gesperrt bleiben, da bei einem Split des Wurzelknotens ein neuer Wurzelknoten entsteht und dadurch eine Aktualisierung im Katalog notwendig wird. Wenn in der Wurzelseite genug Platz vorhanden ist, um einen maximalen Datensatz aufzunehmen, kann die Katalogseite wieder entsperrt werden, da nun ein Split von der Wurzelseite aufgefangen werden kann, ohne eine neue Wurzelseite zu erzeugen. Wie in Kapitel 4 beschrieben, wird nun die Suche in einer Sohnseite der Wurzel fortgesetzt. Bei jedem Besuch einer neuen Seite wird geprüft, ob diese Seite einen maximalen Datensatz aufnehmen kann. Vorausgesetzt es ist ausreichend Platz vorhanden, können alle bis dahin gesperrten Seiten entsperrt werden. Dadurch ist es möglich, dass nun eine zweite Transaktion auf diese Seiten zugreifen kann, bevor die erste Transaktion vollständig beendet ist.

Beim Löschen eines Schlüssels aus einem B-Baum kann es zu einem Unterlauf kommen, wenn die Seite, in der sich der Schlüssel befindet, nach dem Löschen des Schlüssels weniger als halb belegt ist. Wird ein Datensatz in einer inneren Seite gelöscht, wird an diese Position ein neuer Schlüssel eingefügt, normalerweise der nächst größere Schlüssel. Tritt der Fall ein, dass der gelöschte Schlüssel sehr klein ist, der Schlüssel, der an diese Position kopiert wird, jedoch sehr groß ist, so kann es auch beim Löschen zu einem Überlauf einer Seite kommen. Deswegen können bei einem Löschvorgang nur dann die Elternseiten freigegeben werden, wenn in der aktuell betrachteten Seite genügend Platz frei ist, sowohl einen Schlüssel zu löschen, ohne einen Unterlauf zu verursachen, als auch einen Schlüssel einzufügen, ohne einen Überlauf auszulösen.

Bei der Suche eines Schlüssels können die Elternseiten immer sofort entsperrt werden, da bei Lesetransaktionen keine Änderungen auftreten können.

### Implementierungsdetails zum vorzeitigen Entsperren

Zuerst wird anhand der Operation, die ausgeführt werden soll, entschieden, welche Sperren nötig sind:

Beim Aufruf von *getFromIndex*() werden alle Seiten im *read*-Modus gesperrt. Dadurch können andere Lesetransaktionen parallel auf die Seiten zugreifen. Beim Aufruf von *deleteFromIndex(), insertIntoIndex()* und *updateIndex()* werden alle Seiten im *update*-Modus gesperrt, da mit Änderungsoperationen zu rechnen ist.

Bei allen Operationen muss sowohl beim B-Baum als auch beim B*-Baum eine Schlüsselposition im B-Baum gesucht werden. Beim B-Baum ist das selbstverständlich, weil er die Datensätze enthält. Beim B*-Baum indexiert ein B-Baum eine Liste, die die eigentlichen Schlüssel enthält. Trotzdem ist die Suchstrategie im B-Baum bei beiden Indextypen identisch. Deswegen wird bei fast allen Operationen bei beiden Indextypen die Methode *getBTreeRecordLocationForKeyLessEqual()* aufgerufen. Die einzige Ausnahme ist die Funktion *insertIntoIndex()* beim B-Baum, da beim B-Baum auch Datensätze als Wegweiser für Wertebereiche eingesetzt werden und es möglich sein muss, diese an explizite Positionen einzufügen. Außerdem geschieht das Einfügen im B-Baum immer *bottom-up*, also von den Blattseiten beginnend. Deswegen muss die Position in einer Blattseite ermittelt werden, was bei *getBTreeRecordLocationForKeyLessEqual()* nicht sichergestellt werden kann.

Bei einem B-Baum-Index hat man nach Aufruf der Methode *getBTreeRecordLocationForKeyLessEqual()* die gesuchte Schlüsselposition unabhängig von der Operation ermittelt. Bei *getFromIndex(), deleteFromIndex()* und *updateIndex()* existiert der Schlüssel bereits, der durch die Methode *getBTreeRecordLocationForKeyLessEqual()* gefunden wird, und es kann abhängig von der gewünschten Operation der Wert ausgegeben werden, der Datensatz gelöscht werden oder der Wert geändert werden.

Bei einem B*-Baum hat man nach Aufruf dieser Methode erst die Seite ermittelt, in welcher der gesuchte Datensatz in der Liste liegen muss, wenn er existiert. In dieser Seite wird dann durch sequentielle Suche der gewünschte Datensatz bzw. die Einfügeposition gesucht.

Das vorzeitige Entsperren geschieht in der Methode *getBTreeRecordLocationForKeyLess-*

*Equal()*. Bei jedem Aufrufen einer neuen Seite wird die Pufferposition dieser Seite in eine *ArrayList* gespeichert. In dieser *ArrayList* stehen also die besuchten (und damit gesperrten) Seiten in korrekter Reihenfolge. Abhängig von der Operation und dem benutzten Speicherplatz der Seite können mit Hilfe dieser *ArrayList* alle anderen Seiten freigegeben werden.

## 4.6 Speicherplatzverbrauch

### 4.6.1 Speicherplatzverbrauch im Dokumentcontainer

Kapitel 2.3 beschreibt, wie ein Dokument in Dokumentenordnung im *Dokumentcontainer* gespeichert wird. Diese Ordnung lässt sich hervorragend zur Präfix-Komprimierung nutzen, da ein Sohn stets seinen Elternknoten als Präfix hat und Geschwisterknoten den gleichen Elternknoten, also dementsprechend immer ein gemeinsames Präfix besitzen. Dies veranschaulicht Tabelle 4.2. Bei dieser Tabelle werden das Präfix und der zu speichernde Rest nach Divisions getrennt. In der getesteten Implementierung geschieht dies allerdings auf Byte-Ebene, um auch andere Schlüsselformate zu unterstützen.

**Tabelle 4.2** Präfix-Komprimierung im Dokumentcontainer

| DeweyID | Präfix vom Vorgänger | zu speichernder Rest |
|---|---|---|
| 1 | | 1 |
| 1.3 | 1 | 3 |
| 1.3.1.3 | 1.3 | 1.3 |
| 1.3.1.5 | 1.3.1 | 5 |
| 1.3.3 | 1.3 | 3 |
| 1.3.3.3 | 1.3.3 | 3 |
| 1.3.5 | 1.3 | 5 |
| 1.3.5.3 | 1.3.5 | 3 |

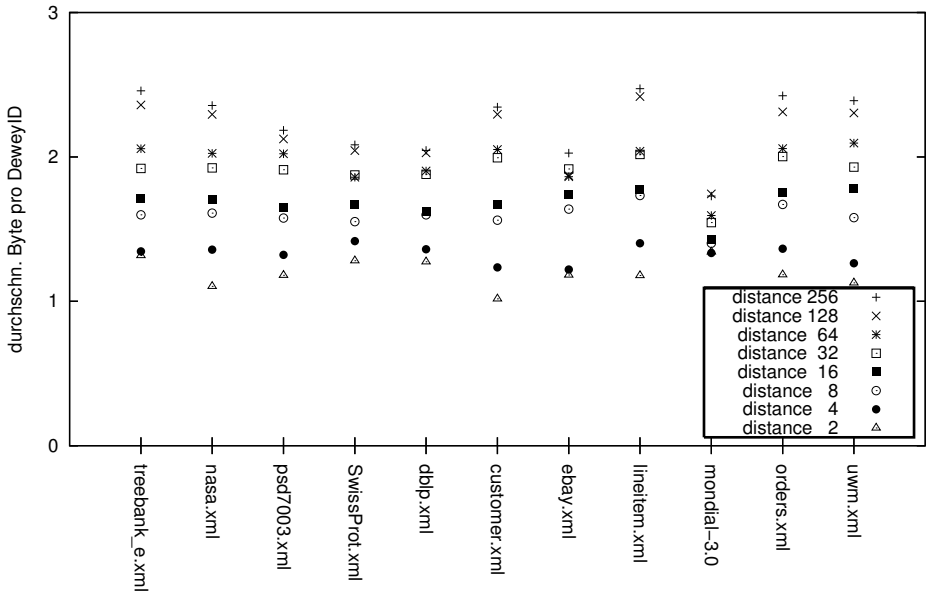
Wie man sieht, muss meist nur die letzte Division gespeichert werden. Eine Ausnahme bildet das erste Attribut eines Knotens, für das noch die Division des virtuellen Wurzelknotens mitgespeichert werden muss.

Um das Potential der Präfix-Komprimierung zu ermitteln, wurde untersucht, wie viele Bytes sich ein Knoten durchschnittlich von seinem Vorgänger unterscheidet, d. h. die Anzahl der Bytes, die gespeichert werden müssen. Bei der Berechnung der folgenden Grafiken sind weitere benötigte Bytes, genauer gesagt die Anzahl der Bytes des Schlüssels sowie die Länge des Präfixes, noch nicht berücksichtigt, da an dieser Stelle mehrere Möglichkeiten bestehen, die Kapitel 2.3 beschreibt.

Abbildung 4.17 zeigt den Speicherplatzverbrauch für alle Testdokumente bei verschiedenen *Distance*-Werten.

Beim Vergleich mit Abbildung 3.2 wird das erhebliche Potential der Präfix-Komprimmierung deutlich. Während ohne Präfix-Komprimierung pro DeweyID zwischen

**Abbildung 4.17** Durchschn. Speicherplatzverbrauch pro DeweyID im *Dokumentcontainer* mit Präfix-Komprimierung




3 und 16 Bytes benötigt werden, reichen bei durchgeführter Präfix-Komprimierung zwischen 1 und 2,5 Bytes aus, wobei dort noch eine nicht näher spezifizierte Längenangabe zur Bestimmung des Präfixes der vorherigen DeweyID hinzukommt. Damit spart man bei der Präfix-Komprimierung etwa zwischen 70% und 85 % des Speicherplatzes ein.

### 4.6.2 Speicherplatzverbrauch im Elementindex

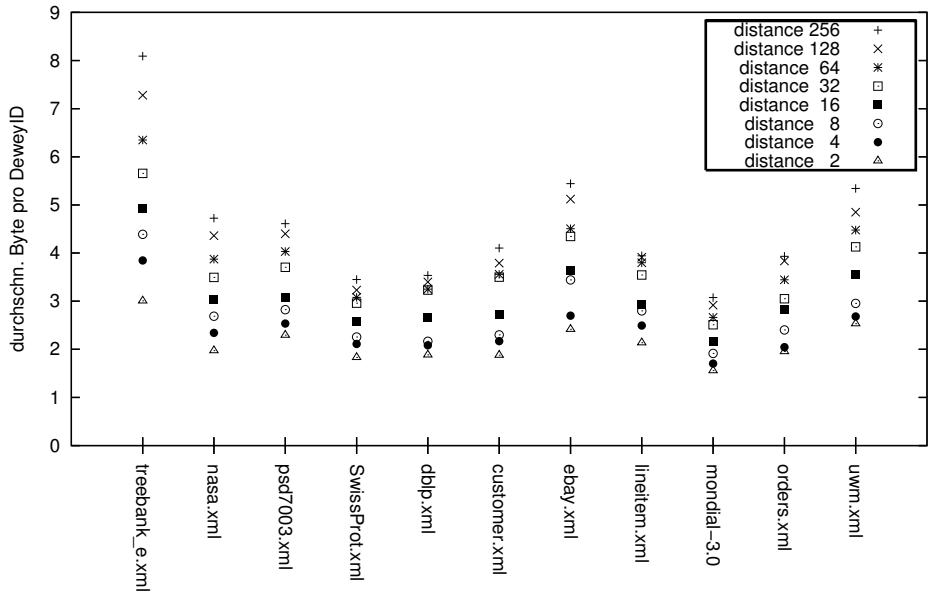
Auch hier wurde das Potential der Präfix-Komprimierung nach der gleichen Vorgehensweise wie zuvor bei dem *Dokumentcontainer* getestet und untersucht, um wie viele Bytes sich eine DeweyID vom Vorgänger unterscheidet. Das Ergebnis ist in Abbildung 4.18 dargestellt.

Auch hier ergibt sich durch die Verwendung der Präfix-Komprimierung ein großes Einsparpotential, das allerdings nicht so hoch ist wie im *Dokumentcontainer*. Im Elementindex wird etwa 50 % Speicherplatz eingespart. Die Ursache für die geringere Effizienz der Präfix-Komprimierung wird klar, wenn man Tabelle 4.3 ansieht. In ihr wird der *Knotenindex* für das Element *last* aus dem XML-Baum in Abbildung 2.3 dargestellt.

**Tabelle 4.3** Präfix-Komprimierung im *Knotenindex* für das Element *last*

| DeweyID | Präfix vom Vorgänger | zu speichernder Rest |
|---|---|---|
| 1.3.5.3 | | 1.3.5.3 |
| 1.5.5.3 | 1 | 5.5.3 |
| 1.5.7.3 | 1.5 | 7.3 |
| 1.5.9.3 | 1.5 | 9.3 |
| 1.7.5.3 | 1 | 7.5.3 |

**Abbildung 4.18** Durchschn. Speicherplatzverbrauch pro DeweyID im Elementindex mit Präfix-Komprimierung




Im Vergleich zu Tabelle 4.2 erkennt man, dass der zu speichernde Rest im Elementindex deutlich größer ist als im *Dokumentcontainer*. Die zweite DeweyID hat mit der ersten nur die erste Division als gemeinsames Präfix und muss folglich drei Divisions speichern. Die dritte und vierte DeweyID können zwar die 1.5 der vorigen DeweyID verwenden, müssen aber immer noch jeweils zwei Divisions (7.3 bzw. 9.3) selbst speichern. Die letzte DeweyID muss wieder drei Divisions speichern (7.5.3). Obwohl die Präfix-Komprimierung auf Byte-Ebene und nicht auf Division-Ebene abläuft, wird deutlich, dass vor allem Elemente, die auf tiefen Ebenen liegen, wenig Nutzen aus der Präfix-Komprimierung ziehen können, da sie nur die gemeinsamen Vorfahren nutzen können. Trotzdem ist das Einsparpotential auch hier beachtlich.

### 4.6.3 Optimierte Kodierungstabellen zur Präfix-Komprimierung

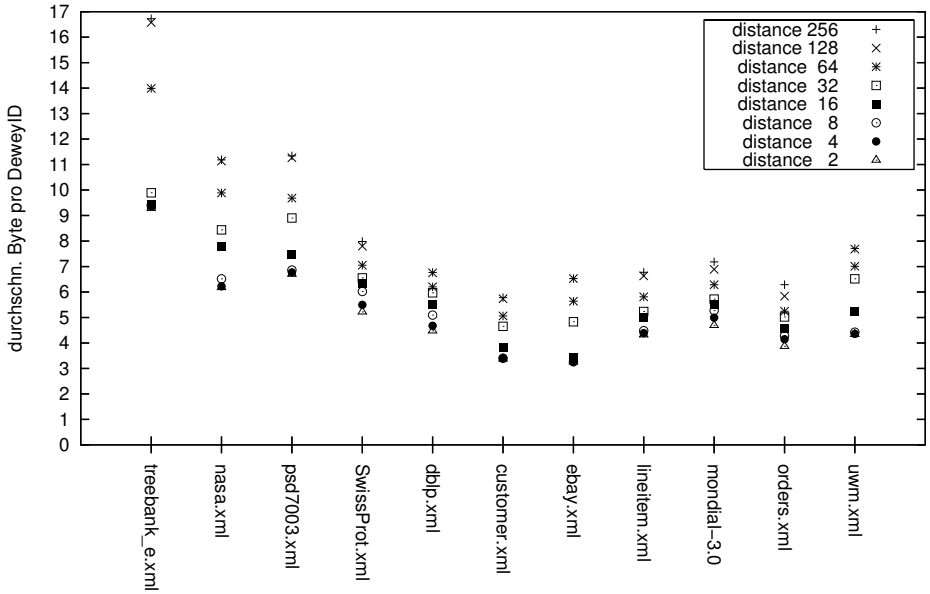
Tabelle 3.2 zur Kodierung der Divisions dient nur als Beispiel. Wie dieses Kapitel zeigen wird, kann eine bessere Kodierung den Speicherplatzverbrauch insbesondere bei der Präfix-Komprimierung weiter verbessern. Wie bereits in Kapitel 3.3 beschrieben, findet die Präfix-Komprimierung auf Byte-Ebene statt. Bei Kodierung mit Tabelle 3.2 tritt der negative Effekt auf, dass eine zusätzliche Division oft mehr Bytes ändert als unbedingt nötig. In Kapitel 3.2.3 wurde gezeigt, wie die DeweyID 1.7.27 kodiert wird. Sie lautet 01111010.00011, wobei die fehlenden drei Bits des zweiten Bytes mit 0 gefüllt werden, d. h., es wird effektiv 01111010.00011000 gespeichert. Ein Sohn des Knotens 1.7.27 könnte 1.7.27.5 sein. Die Division 5 wird nach Tabelle 3.2 durch 0101 kodiert. Die DeweyID 1.7.27.5 wird daher mit 01111101.00001101.01 kodiert bzw. mit 01111101.00001101.01000000 gespeichert. Bei der Präfix-Komprimierung müssen also das zweite und dritte Byte gespeichert werden, da die zusätzliche Division 5 die Byte-Grenze überschreitet und dadurch beide Byte-Werte ändert. Um diesen Effekt zu verhindern, wurde der Speicherplatzverbrauch wie in Kapitel 3.3

mit der Kodierungstabelle 1 aus Tabelle 3.7 erneut untersucht. Bei dieser Kodierungstabelle, im Folgenden Kodierungstabelle K1 genannt, sind die Wertebereiche an die Byte-Grenzen angepasst.

### 4.6.3.1 Speicherplatzverbrauch im *Dokumentcontainer*

Abbildung 4.19 zeigt den unkomprimierten Speicherplatzverbrauch im *Dokumentcontainer*.

**Abbildung 4.19** Speicherplatzverbrauch mit K1 im *Dokumentcontainer* ohne Präfix-Komprimierung



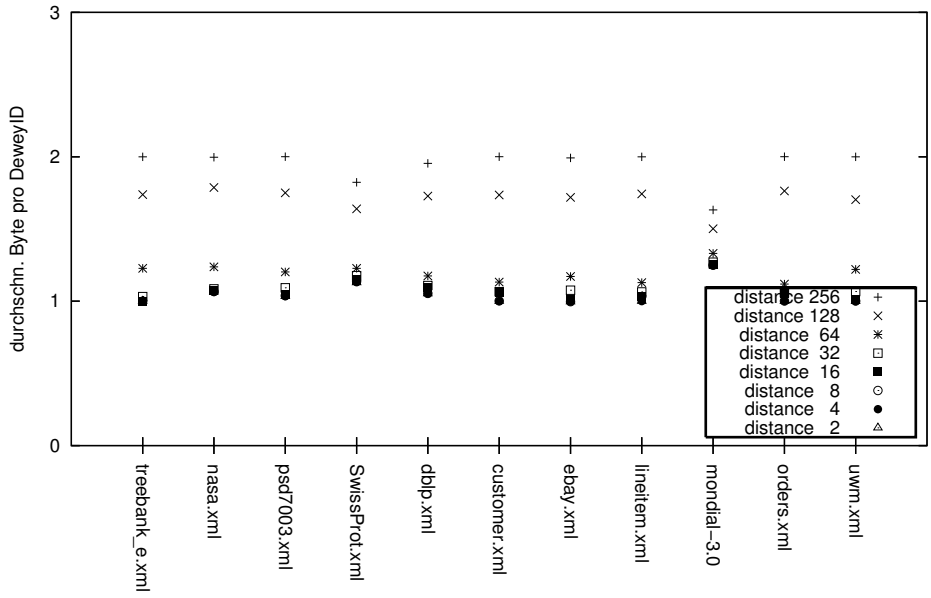

Der Speicherplatzverbrauch liegt im Gegensatz zum Speicherplatzverbrauch aus Abbildung 3.2, die die ursprüngliche Kodierungstabelle verwendet, vor allem bei einem kleinen *Distance*-Wert etwas höher. Das ist zu erwarten, da jetzt vor allem kleine Division-Werte mehr Speicherplatz benötigen. Bei den mittleren *Distance*-Werten dagegen ist der Speicherplatz der neuen Tabelle besser, da hier mehr Division-Werte zwischen 24 und 127 vorkommen, die mit weniger Speicherplatz kodiert werden können.

In Abbildung 4.20 ist der Speicherplatzverbrauch im *Dokumentcontainer* mit Präfix-Komprimierung abgebildet. Hier ist der bereits niedrige Speicherplatzverbrauch, den Abbildung 4.17 zeigt, nochmals deutlich gesunken. Der durchschnittliche Speicherplatzverbrauch pro DeweyID erreicht nur bei *Distance*-Wert 256 die 2 Byte-Grenze. Bei *Distance*-Werten kleiner als 128 werden maximal 1,33 Bytes (*Distance*-Wert 64 bei *mondial.xml*) pro DeweyID benötigt. Betrachtet man noch kleinere *Distance*-Werte, bewegt sich der Speicherplatzverbrauch sehr nahe bei 1 Byte (1,0001 Bytes bei *treebank.xml*), was bei diesem Verfahren den theoretisch minimalen Wert darstellt.

### 4.6.3.2 Speicherplatzverbrauch im Elementindex

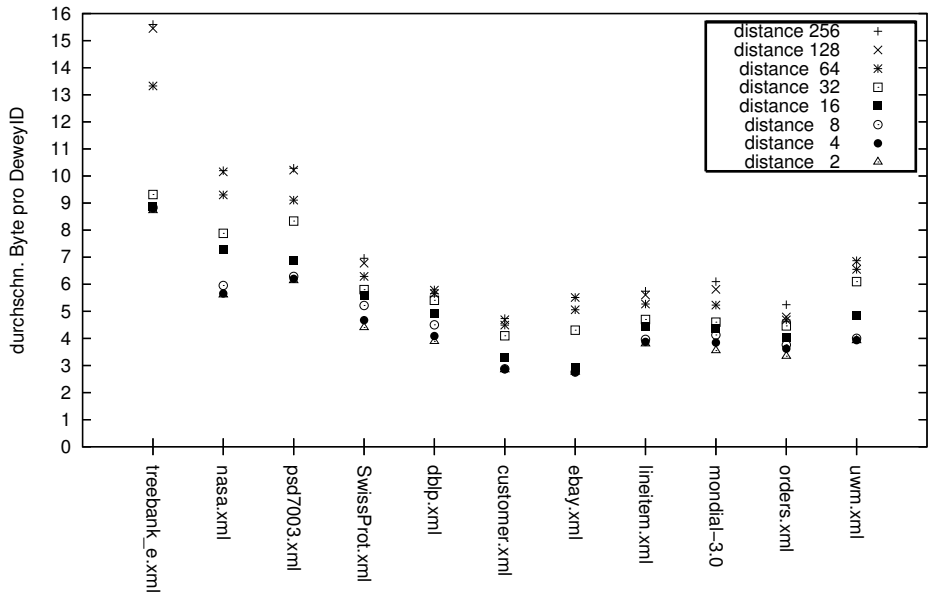
Abbildung 4.21 zeigt den Speicherplatzverbrauch im Elementindex. Hier liegt der Speicherplatzverbrauch im Vergleich zu Abbildung 3.3 bei geringen Distance-Werten wie beim *Do-*

**Abbildung 4.20** Speicherplatzverbrauch mit K1 im *Dokumentcontainer* mit Präfix-Komprimierung




*kumentcontainer* etwas höher, während im Mittelfeld der *Distance*-Werte diese Kodierung meist effizienter ist.

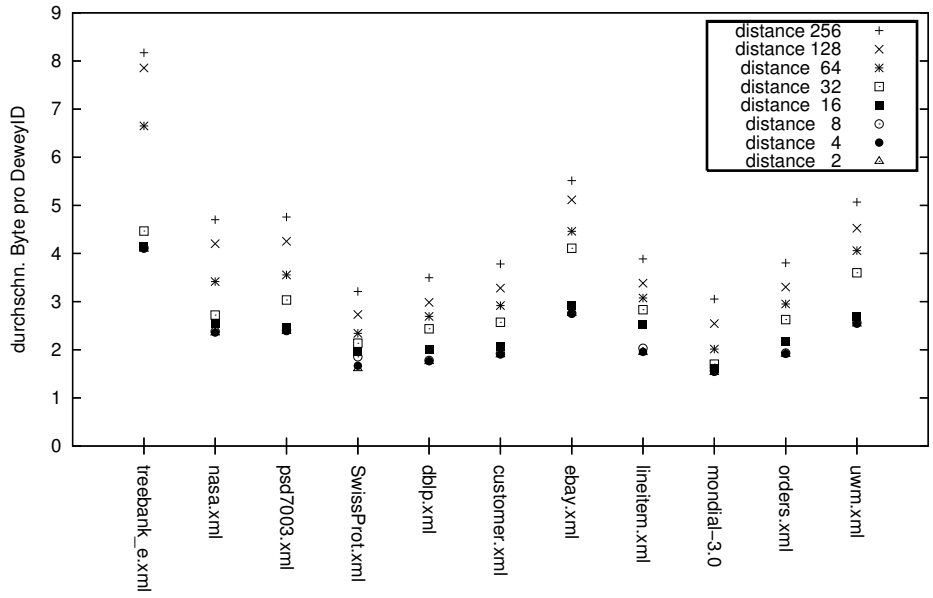
**Abbildung 4.21** Speicherplatzverbrauch mit K1 im Elementindex ohne Präfix-Komprimierung




Im Elementindex mit Präfix-Komprimierung ist die Kodierung, die an Byte-Grenzen angepasst ist, in den meisten Fällen etwas besser (siehe Abbildung 4.22 im Vergleich zu Abbildung 4.18), wobei hier der Unterschied nicht so stark ausfällt wie im *Dokumentcontainer*. Allerdings kann man in Abbildung 4.22 auch erkennen, dass der Speicherplatzverbrauch bei Variation der *Distance* von 2 bis 32 nicht so stark ansteigt wie bei Abbildung 4.18. Aus

diesem Grund ist Kodierungstabelle 1 im Bereich zwischen *Distance* 8 und 32 besser. Dagegen ist bei *treebank_e.xml*, *nasa.xml*, *psd7003.xml* und *ebay.xml* bei *Distance* 2 noch die ursprüngliche Kodierungstabelle (Tabelle 3.2) besser.

**Abbildung 4.22** Speicherplatzverbrauch mit K1 im Elementindex mit Präfix-Komprimierung




### 4.6.3.3 Weitere Kodierungstabellen

Es gibt weitere Möglichkeiten, Kodierungstabellen zu erstellen, bei denen die einzelnen Wertebereiche an Byte-Grenzen angepasst sind. Diese und Variationen werden in diesem Kapitel untersucht. Die Variationen entstehen aus der Beobachtung, dass bei der Kodierungstabelle 1 das erste Attribut eines Elements immer zwei Bytes verwendet, da es sich von seinem Elementknoten durch die zwei Divisions 1.3 unterscheidet und beide Divisions gespeichert werden müssen. Die 1 des Attributwurzelknotens wird vom ersten Attribut gespeichert, da der Attributwurzelknoten selbst nicht gespeichert wird. Deshalb wurden auch Kodierungsmöglichkeiten untersucht, bei denen die Forderung nach strikten Byte-Grenzen aufgeweicht wurde, und der erste Bereich nur aus vier Bits besteht. So wird die 1.3 mit nur einem Byte kodiert. Allerdings kann dies bei anderen Knoten die Kodierung wieder verschlechtern. Die Verschlechterung tritt ein, falls DeweyIDs eine ungerade Anzahl an Divisions besitzen, die in den Bereich 1-7 fallen (wobei erst ab der zweiten Division gezählt wird, da die erste Division nicht explizit kodiert wird), und eine Division folgt, die nicht in diesen Bereich fällt. Zum Beispiel benötigt die Speicherung der DeweyID 1.17.3 mit Kodierungstabelle 2 (aus Tabelle 3.7) im letzten Byte nur vier Bits. Wenn als nächste DeweyID aber 1.17.3.9 gespeichert wird, überschreitet die Division 9, die acht Bits benötigt, eine Byte-Grenze, und es müssen zwei Bytes gespeichert werden.

Wenn man wie bei Kodierungstabelle 1 fünf verschiedene Wertebereiche verwendet und zusätzlich erlaubt, dass der erste Wertebereich nur vier Bits verwendet, ergeben sich zusätzlich zu Kodierungstabelle 1 die in Abbildung 3.7 gezeigten Tabellen.

Es besteht außerdem die Möglichkeit, nur vier Wertebereiche zu benutzen. Die dabei möglichen Kodierungstabellen sind in Abbildung 3.8 gezeigt.

Für alle Kodierungstabellen wurden Auswertungen bezüglich des Speicherplatzverbrauchs durchgeführt und die Ergebnisse miteinander verglichen.

Beim Vergleich von Kodierungstabelle 1 (K1) und Kodierungstabelle 6 (K6) fällt auf, dass K1 und K6 die ersten drei Wertebereiche gleich kodieren, und der Wertebereich vier von K6 bei K1 in zwei Wertebereiche aufgeteilt ist. Daraus folgt, dass Divisions, die in den vierten Wertebereich von K1 fallen, mit K1 effizienter kodiert werden können. Aus diesem Grund ist K1 immer gleich oder besser als K6. Dies bestätigt auch der Vergleich der Auswertungen.

Bei dem Vergleich von K1 mit K7 werden Divisions bis 16511 identisch kodiert. Divisions, die in den dritten Wertebereich von K1 fallen, werden bei K1 effizienter kodiert (drei Bytes) als bei K7 (vier Bytes). Divisions im vierten Wertebereich von K1 werden wie bei K7 mit vier Bytes kodiert. Divisions, die in den dritten Wertebereich von K7 passen, nicht aber in den vierten Wertebereich von K1, werden in K7 effizienter kodiert. Insgesamt kann man beim Vergleich bei der Präfix-Komprimierung der beiden Auswertungen keine Unterschiede feststellen. In der unkomprimierten Version besitzt allerdings K1 aufgrund der besseren Kodierung im dritten Wertebereich einen Vorteil, wodurch die maximale Länge der unkomprimierten DeweyID bei K1 meist besser ist. Aus diesem Grund wird K1 bevorzugt.

Bei dem Vergleich der Wertetabellen K1 und K8 stellt man fest, dass der zweite Wertebereich in K8 bereits drei Bytes verwendet. Deswegen werden Divisions, die größer sind als 128, im Vergleich zu K1 sehr schlecht kodiert. Das spiegelt sich auch in den Auswertungen wider, in denen K8 deutlich schlechter abschneidet. Je höher der *Distance*-Wert ist, desto schlechter schneidet dabei K8 ab.

Damit geht K1 eindeutig als Sieger im Vergleich zu Kodierungstabellen hervor, die sich strikt an die Byte-Grenzen halten. Als nächstes wurden die Kodierungstabellen untereinander verglichen, bei denen der erste Wertebereich nur vier Bits benötigt. Diese Kodierungstabellen haben das Ziel, bei Dokumenten mit Attributen besser abzuschneiden.

Vergleicht man K2 und K3, so eignet sich K3 besser zur Kodierung, da bei K2 die Divisions, die nicht in den vierten Wertebereich passen, sehr ineffizient kodiert werden. Diese Grenze liegt bereits bei Division-Wert 1,05 Millionen, den auch Dokumente mittlerer Größe schnell erreichen. Ein Dokument mit einem Fanout von 20.000 und *Distance*-Wert 50 erreicht beispielsweise bereits den Division-Wert 1 Millionen.

Bei K4 werden bereits ab dem Division-Wert 72 drei Bytes zur Kodierung benötigt. Divisions, die in den Bereich zwischen 72 und 8263 fallen, werden deswegen von K3 besser kodiert. Da in diesem Bereich relativ viele Divisions verwendet werden, fällt das Ergebnis eindeutig zugunsten von K3 aus.

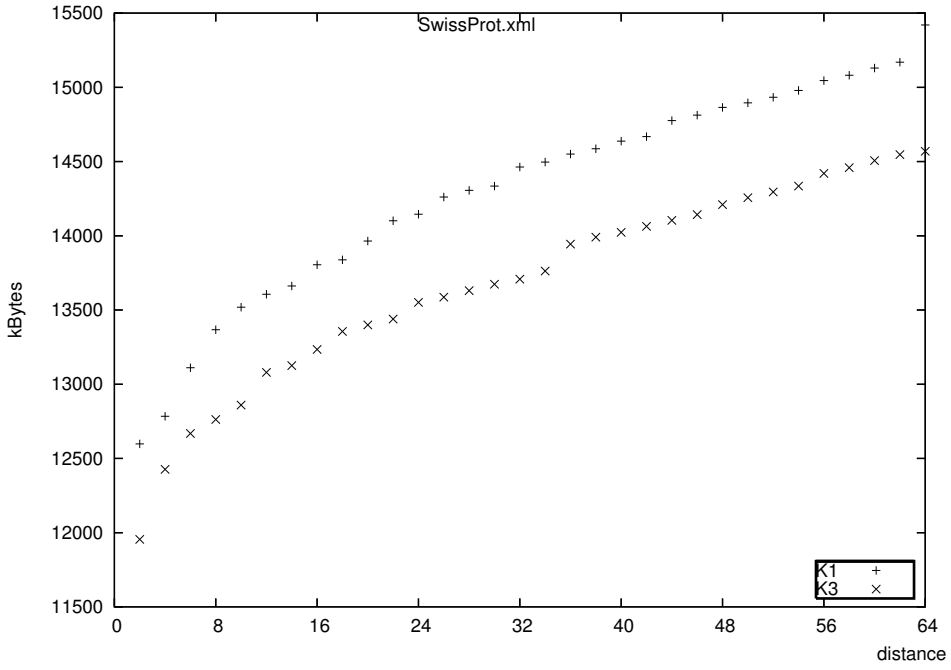
K5 schneidet im Vergleich zu K3 gleichermaßen sehr schlecht ab, da bereits bei den sehr häufig vorkommenden Division-Werten 8-71 zwei Bytes verwendet werden müssen, bei denen in K3 noch ein Byte ausreicht.

Beim Vergleich zwischen K3 und den Kodierungstabellen, die nur vier Wertebereiche un-

terscheiden (K9, K10, K11, K12, K13), schneidet K3 ebenfalls besser ab, da fünf Wertebereiche eine feinere Unterscheidung zulassen, von der gerade kleine Division-Werte profitieren. Bei den Kodierungstabellen mit nur vier Wertebereichen werden zwar manche hohe Division-Werte besser kodiert, jedoch auf Kosten niedriger Division-Werte. Zum Beispiel wird bei K11 der Bereich 8264-16391 um ein Byte besser kodiert als bei K3, was jedoch auf Kosten der Werte zwischen 8-71 geht, die deutlich häufiger vorkommen.

Zum Schluss müssen noch K3 und K1 verglichen werden. Hier ergibt sich kein eindeutiges Bild, da von der Struktur der Dokumente und dem gewählten *Distance*-Wert abhängt, welche Kodierung besser ist. In Abbildung 4.23 und 4.24 zeigen die Beispiele *Swissprot.xml* und *mondial-3.0.xml*, dass die Idee funktioniert, erste Attribut besser zu kodieren, wenn der erste Wertebereich nur vier Bits benötigt. Um bei diesen Grafiken sowohl *Dokumentcontainer* als auch Elementindex zusammen zu berücksichtigen, wird in diesen wie in den folgenden Grafiken die Summe des Speicherplatzverbrauchs aller DeweyIDs aus *Dokumentcontainer* und Elementindex dargestellt.

**Abbildung 4.23** Speicherplatzverbrauch von *Swissprot.xml* (K1 im Vergleich zu K3)




Der Speicherplatzverbrauch von K3 sinkt im Gegensatz zu K1 bei beiden Dokumenten für alle *Distance*-Werte deutlich. Abbildung 4.25 veranschaulicht, dass dies nicht immer so sein muss. Obwohl *dblp.xml* eine hohe Anzahl von Attributen besitzt, ist K3 nur bei *Distance*-Werten von 2 bis 6 besser. Danach ist bis *Distance* 32 K1 besser. Das liegt daran, dass bei *dblp.xml* ab *Distance*-Wert 8 die Anzahl der Divisions, die über 71 liegen, stark ansteigt. Außerdem kann K1 die Divisions von 72 bis 127 noch in einem Byte darstellen, während bei K3 zwei Bytes verwendet werden müssen. Da ab *Distance* 34 die meisten Divisions größer als 127 werden, ist von da an der Speicherplatzverbrauch etwa gleich.

Es bleibt zu klären, wie gut K3 bei Dokumenten ohne oder mit nur wenigen Attributen

**Abbildung 4.24** Speicherplatzverbrauch von *mondial-3.0.xml* (K1 im Vergleich zu K3)

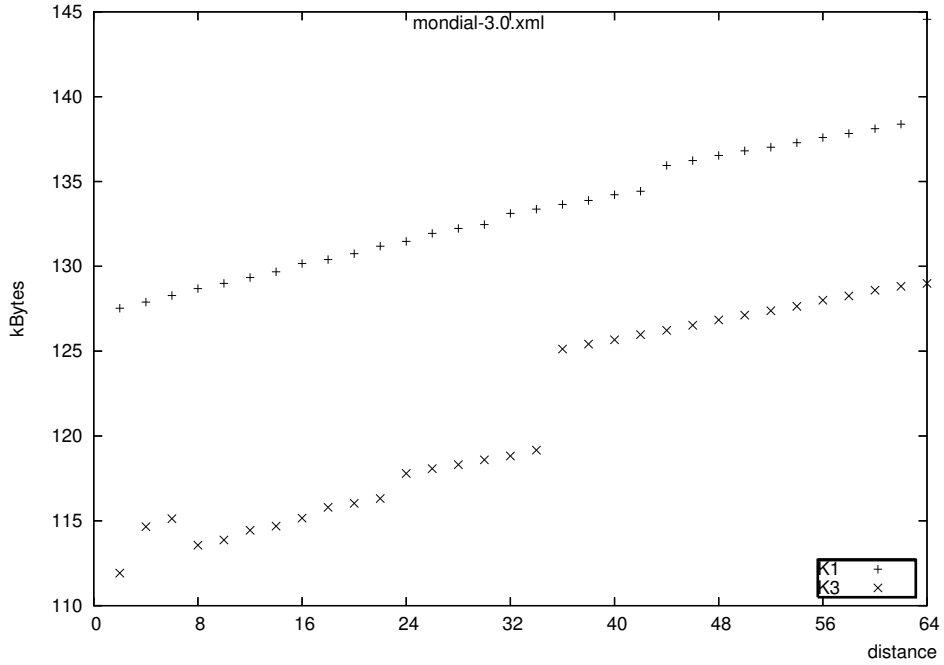
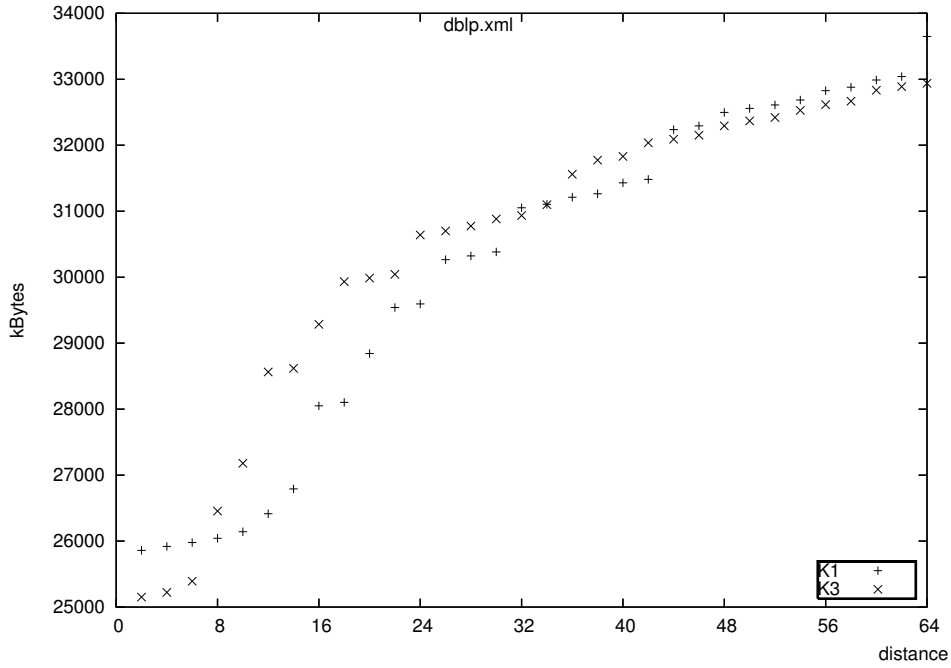



**Abbildung 4.25** Speicherplatzverbrauch von *dblp.xml* (K1 im Vergleich zu K3)



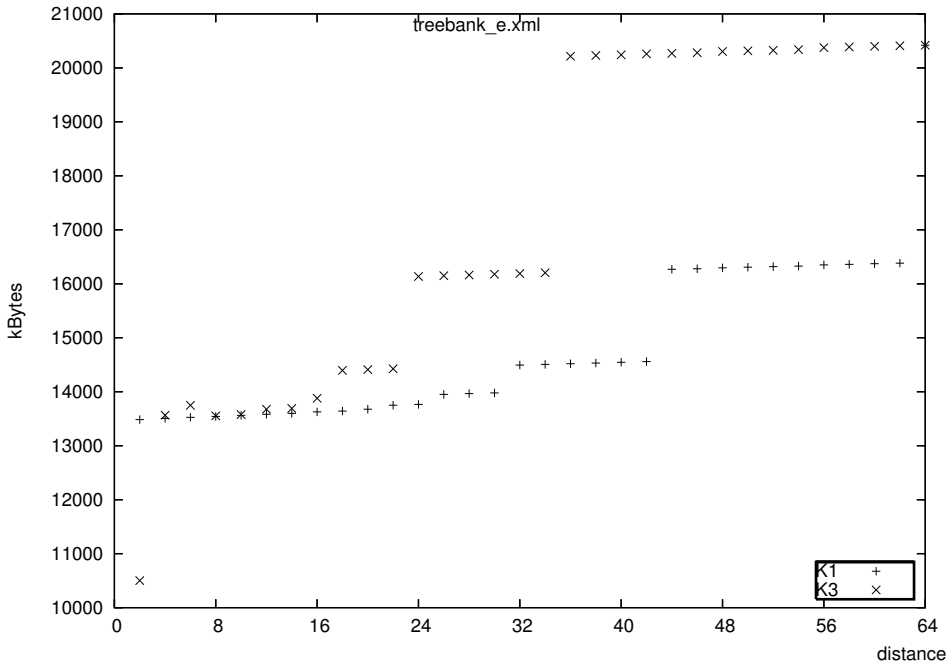

abschneidet. Als Beispiel dafür ist in den Abbildungen 4.26 und 4.27 der Speicherplatzverbrauch der Dokumente *treebank_e.xml* und *nasa.xml* für die Kodierungstabellen K1 und K3 abgebildet. Bei beiden Dokumenten eignet sich K3 bei *Distance*-Wert 2 deutlich besser. Dieser Vorteil entsteht nur durch den Elementindex, da im Elementindex die Anpassung der Byte-Grenzen keine so große Rolle spielt, und es wichtiger ist, die unkomprimierten

DeweyIDs möglichst klein zu halten, damit der zu speichernde Rest möglichst wenig Speicherplatz verbraucht.

Bei *Distance*-Wert 4 ist der Speicherplatzverbrauch von K3 bei *treebank_e.xml* bereits höher als der von K1 und bei *nasa.xml* schon fast identisch. Spätestens bei *Distance*-Wert 8 ist der Speicherplatzverbrauch von K1 gleich oder besser. Die Grafiken der anderen Dokumente werden in Abbildung 4.28 dargestellt. Bei *ebay.xml* hat K3 nur bei *Distance*-Wert 2 einen besseren Speicherplatzverbrauch und bei *psd7003.xml* bis *Distance*-Wert 8. Bei den restlichen Dokumenten ist der Speicherplatzverbrauch entweder identisch mit K1 oder K1 ist besser.

Insgesamt lässt sich sagen, dass K3 bei *mondial.xml* und *Swissprot.xml* eindeutig besser abschneidet. Bei allen anderen Dokumenten ist K3 aber nur bei sehr geringen *Distance*-Werten besser. Während K3 bei *Distance*-Wert 2 noch gut abschneidet, ist bereits bei *Distance*-Wert 4 bis 8 meist K1 besser, da K1 bei steigendem *Distance*-Wert besser skaliert.

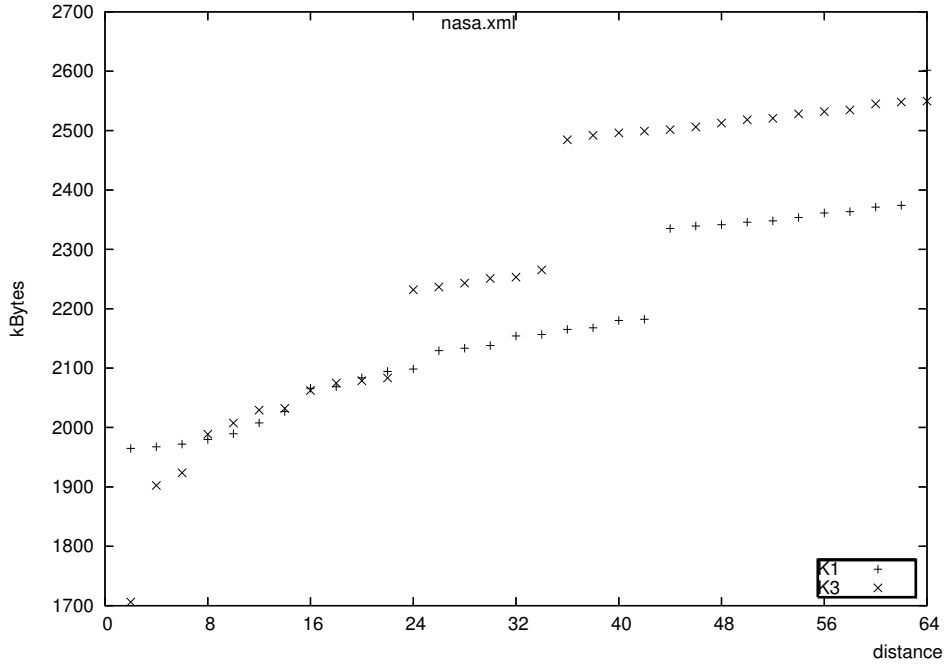
**Abbildung 4.26** Speicherplatzverbrauch von *treebank_e.xml* (K1 im Vergleich zu K3)




Um eine Entscheidung über die Auswahl einer Kodierungstabelle treffen zu können, ist es sinnvoll, auch die maximale DeweyID-Länge der Kodierungen mit einzubeziehen, da sie die Anzahl der Knotenneunummerierungen entscheidend beeinflusst (Kapitel 3.5). Abbildung 4.29 zeigt, dass K3 zwar bei *Distance*-Wert 2 für *treebank_e.xml* deutlich besser abschneidet als K1, die maximale DeweyID-Länge dann aber sofort stark ansteigt und schließlich deutlich schlechter wird als K1. Die im Anhang B enthaltenen Diagramme für die anderen Dokumente zeigen ein ähnliches Bild. Bis auf wenige Ausnahmen ist die maximale DeweyID-Länge von K1 gleich oder besser als K3.

Während K3 nur bei sehr geringen *Distance*-Werten sinnvoll ist, da die Ergebnisse bei wachsenden *Distance*-Werten schnell schlechter werden, eignet sich Kodierungstabelle K1 insge-

**Abbildung 4.27** Speicherplatzverbrauch von *nasa.xml* (K1 im Vergleich zu K3)



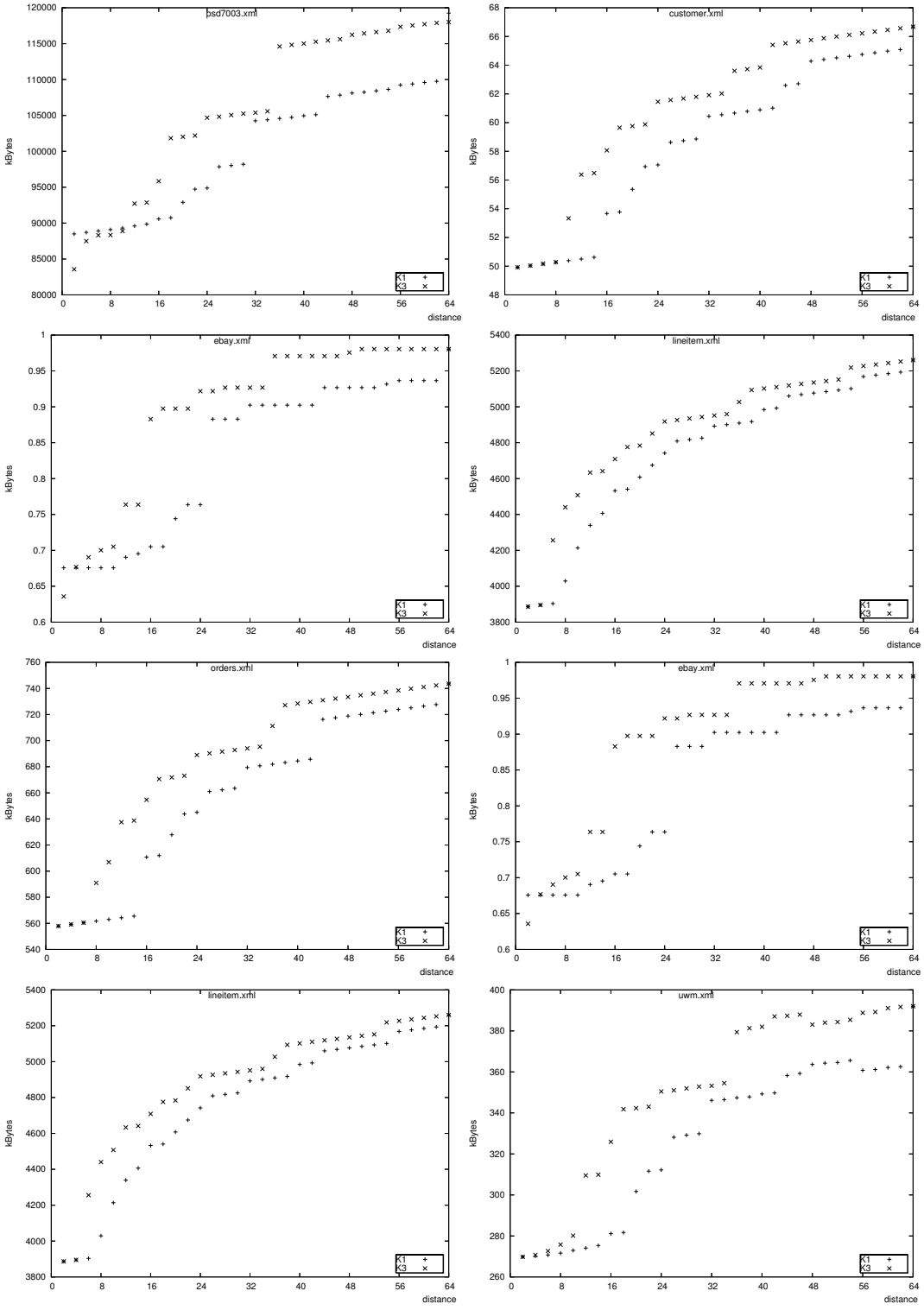

samt besser, da mit ihr auch bei größeren *Distance*-Werten der Speicherplatzverbrauch gut skaliert. Aus diesen Gründen wird K1 zur Kodierung verwendet.

### 4.6.4 CutOff-Länge

Kapitel 4.4.2 beschreibt das Format zur Speicherung von Schlüssel/Wert-Paaren beider Versionen.

Da die CutOff-Länge und die DiffKey-Länge meist sehr klein sind, wird standardmäßig nur ein Byte für beide Werte zusammen verwendet. Die ersten vier Bits sind für die CutOff-Länge reserviert und die letzten vier Bits für die DiffKey-Länge. Mit diesen vier Bits werden jeweils die Werte 0 bis 14 dargestellt. Muss ein Wert größer als 14 dargestellt werden, wird der Wert in der CutOff-Länge bzw. DiffKey-Länge auf 15 gesetzt und zusätzlich die optionale CutOff-Länge bzw. optionale DiffKey-Länge verwendet. Diese beinhaltet dann einen Wert bis 255. Da eine Division kodiert maximal 35 Bits, also fünf Bytes benötigt (bei Kodierungstabelle 1), kann die DiffKey-Länge im *Dokumentcontainer* aufgrund der geordneten Reihenfolge tatsächlich nie größer als 5 sein, weil immer nur die letzte Division gespeichert werden muss. Die restlichen Divisions können vom Elternknoten oder Geschwisterknoten übernommen werden. Bei der Kodierung der DeweyIDs wird das optionale DiffKey-Byte im *Dokumentcontainer* folglich nie verwendet. Die CutOff-Länge variiert dagegen abhängig von der Tiefe des Dokuments. Wenn ein Knoten auf einer sehr tiefen Ebene liegt (z. B. 1.3.5.7.3.5.3.3.3.5) und der Nachfolgeknoten auf einer flachen Ebene (z. B. 1.3.7), müssen viele Bytes abgeschnitten werden (die Bytes, die die Divisions 5.7.3.5.3.3.3.5 kodieren). Dementsprechend kommt es vor allem bei tiefen Dokumenten zu höheren CutOff-Werten. Eine genauere Untersuchung ergab, dass von den hier betrachteten Dokumenten de facto

**Abbildung 4.28** Speicherplatzverbrauch der restlichen Dokumente (K1 im Vergleich zu K3)




nur das Dokument *treebank_e.xml* CutOff-Werte größer als 14 verwendet (untersucht auf der Kodierungstabelle 1). Alle anderen Dokumente benötigen selbst bei *Distance* 256 das optionale CutOff-Byte nicht.

**Abbildung 4.29** Maximale DeweyID-Länge von *treebank_e.xml* (K1 im Vergleich zu K3)

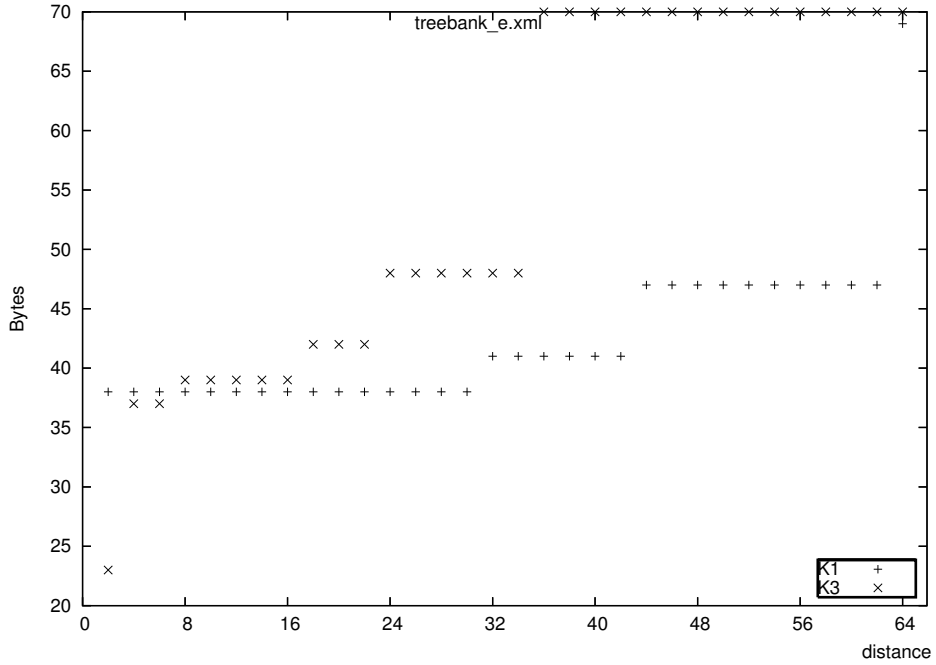
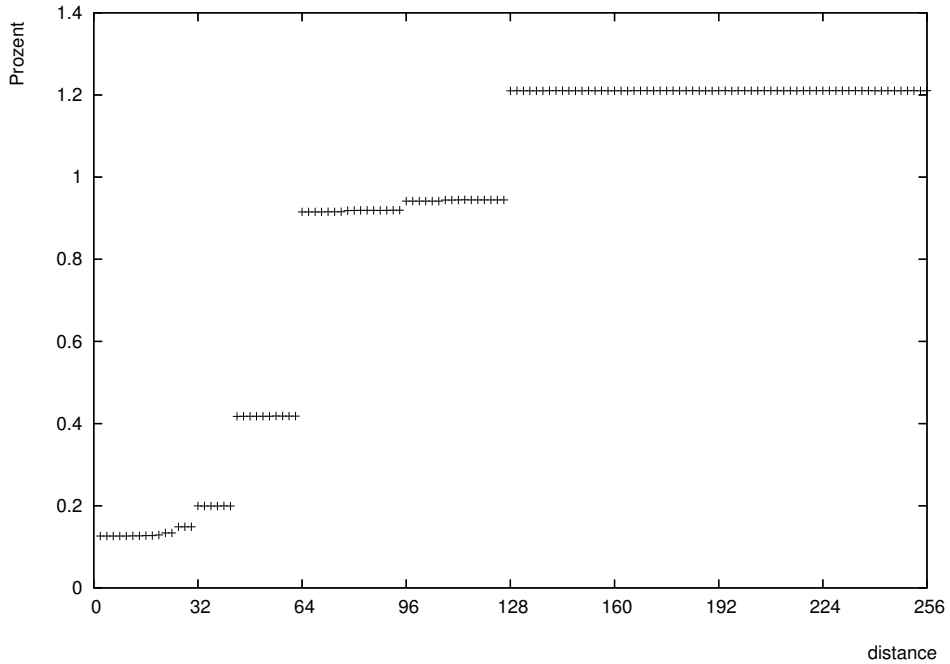



Abbildung 4.30 zeigt in Abhängigkeit der verschiedenen *Distance*-Werte, bei wie viel Prozent der Knoten das optionale Längenfeld-Byte in *treebank_e.xml* verwendet werden muss.

**Abbildung 4.30** Prozentuale Benutzung des optionalen CutOff-Längenfelds im *Dokumentcontainer* in *treebank_e.xml*



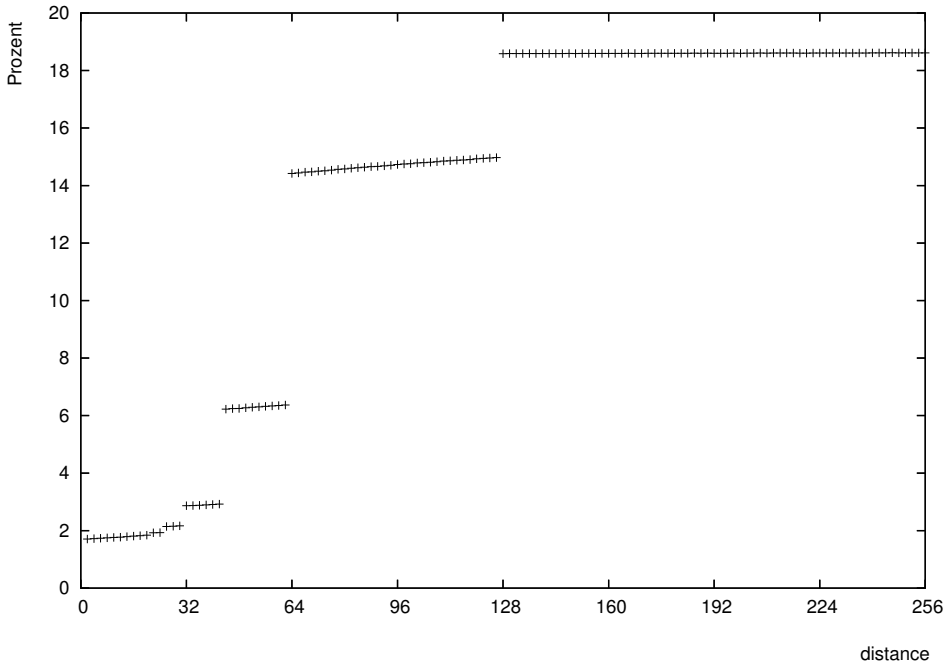

Man sieht, dass das optionale Längenfeld-Byte nur sehr selten benötigt wird. Außer *tree-*

*bank_e.xml* verwendet es keines der anderen untersuchten Dokumente, und selbst bei diesem Dokument benötigen maximal 1,21 Prozent aller Knoten das optionale Längen-Byte.

Im Elementindex wird das optionale CutOff-Längenfeld-Byte öfters benötigt, da bei zwei aufeinander folgenden Elementen auf tiefer Ebene der komplette Teil bis zum gemeinsamen Elternknoten abgeschnitten werden muss. Doch auch hier wird das optionale Längenfeld-Byte nur wenig genutzt. Lediglich *treebank_e.xml* und *nasa.xml* benötigen es im Elementindex. Bei *nasa.xml* ist es jedoch zu vernachlässigen, denn erst ab *Distance* 88 wird es überhaupt verwendet, und selbst bei dem sehr großen *Distance*-Wert 256 nur 21 Mal, was 0.004 Prozent entspricht.

Bei *treebank_e.xml* dagegen steigt die Nutzung stärker an. Das zeigt Abbildung 4.31. Während bei *Distance*-Werten kleiner als 32 nur knapp über 2 Prozent der Knoten das optionale Längenfeld-Byte verwenden, steigt die Zahl der Knoten auf über 18 Prozent bei *Distance*-Werten ab 128 an.

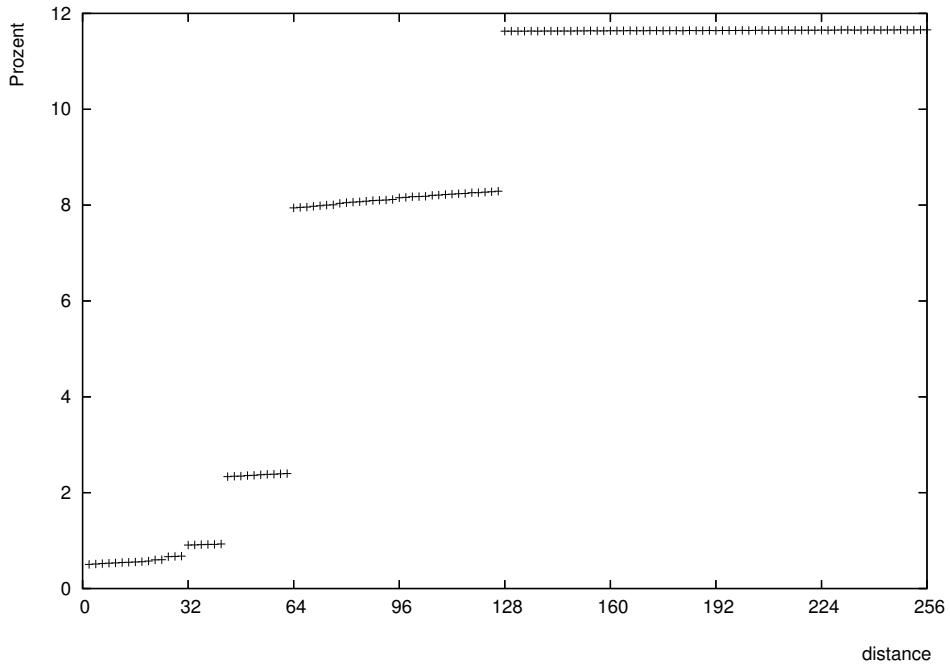
**Abbildung 4.31** Prozentuale Benutzung des optionalen CutOff-Längenfelds im Elementindex in *treebank_e.xml*




Im Elementindex spielt auch die optionale DiffKey-Länge eine Rolle. Sie wird aber ebenfalls nur von *nasa.xml* und *treebank_e.xml* verwendet. Dabei ist sie bei *nasa.xml* erneut zu vernachlässigen, da sie erst bei *Distance*-Wert 128 erstmals auftritt und auch bei *Distance*-Wert 256 nur 18 Mal verwendet werden muss, was weniger als 0.004 Pozent aller Elementknoten entspricht. Der Verbrauch von *treebank_e.xml* ist in Abbildung 4.32 abgebildet. Auch hier ist der zusätzliche Speicherplatzverbrauch, der bei DiffKey-Längen größer als 14 entsteht, insbesondere bei geringen *Distance*-Werten gering.

Die Wertelänge verwendet ebenfalls ein optionales Längenbyte. Standardmäßig wird nur ein Byte benutzt, durch das ein Wert zwischen 0 und 254 definiert werden kann. Soll ein grö-

**Abbildung 4.32** Prozentuale Benutzung des optionalen DiffKey-Längenfelds im Elementindex in *treebank_e.xml*




ßerer Wert dargestellt werden, wird der Wert auf 255 gesetzt und die zwei Bytes „optionale Wertelänge“ verwendet, wodurch bis zu 64 Kilobytes möglich sind. Durch diese variable Speicherungsgröße lässt sich viel Speicherplatz einsparen, da die meisten Datensätze diese optionalen Bytes nicht benötigen.

## 4.7 Leistungsvergleich zwischen Indexmanager 1 und Indexmanager 2

Für den Leistungsvergleich wird das Benchmark-Programm TAMix verwendet, das für den XTC-Server entwickelt wurde. Es simuliert das Datenbanksystem einer Bibliothek. Zum Test wird ein XML-Dokument erstellt, das für die hier verwendeten Tests folgende Parameter benutzt:

- numberOfPersons = 1000
- numberOfAuthors=500
- numberOfTopics=100
- numberOfBooks=2500

Das daraus erstellte Dokument besteht aus etwa 18,6 Megabytes. Die anderen Metriken sind in Tabelle 4.4 aufgelistet.

Dieses Dokument wird in das Datenbanksystem eingelesen und gespeichert. Dann werden mehrere Transaktionen parallel ausgeführt, die sich wie folgt zusammensetzen:

**Tabelle 4.4** Metriken des Test-Dokuments

| Dateiname | *bib.xml* |
|---|---|
| Beschreibung | Bibliotheksdaten |
| Größe (Byte) | 19503380 |
| Anzahl Element-Knoten | 144331 |
| Anzahl der Attribute | 50237 |
| Anzahl der Textknoten | 86990 |
| max Ebene | 7 |
| ∅ Ebene | 5.75 |
| max FanOut | 1000 |
| ∅ FanOut | 1.99 |

- 4 TAqueryPerson
- 5 TAqueryBook
- 16 TAlendAndReturn
- 1 TAqueryAndModifyBook

Die Beschreibung der Transaktionstypen kann in [9] nachgelesen werden.

### 4.7.1 Speicherplatzverbrauch

Es sind mehrere Optimierungen in den Indexmanager eingeflossen, um möglichst viel Speicherplatz einzusparen. Tabelle 4.5 stellt den Speicherplatzverbrauch tabellarisch dar. Indexmanager 1 braucht zur Speicherung des erzeugten Dokuments 29.71 Megabytes, Indexmanager 2 dagegen ohne Präfix-Komprimierung nur 25.77 Megabytes und mit Präfix-Komprimierung 23.29 Megabytes.

Außerdem wurde der Speicherplatzverbrauch analog zu der Vorgehensweise von Kapitel 3.3 untersucht und in Tabelle 4.6 dargestellt. Dabei ergibt sich ein Speicherplatzverbrauch im *Dokumentcontainer* ohne Präfix-Komprimierung von 6.33 Bytes pro DeweyID, das bedeutet 1742 Kilobytes für alle DeweyIDs zusammen. Geht man davon aus, dass die optionalen Bytes im Datensatzformat für den Schlüssel (siehe Kapitel 4.4.2) nicht verwendet werden müssen, erhöht sich der Speicherplatzverbrauch um ein Byte pro DeweyID oder 281558 Bytes insgesamt zur Speicherung der CutOff- und DiffKey-Länge, die in den Auswertungen nicht berücksichtigt wurden. Der rechnerische Speicherplatzverbrauch im *Dokumentcontainer* ist also 1.96 Megabytes.

Im Elementindex ergibt sich bei der Untersuchung des Dokuments eine Größe von insgesamt 795 Kilobytes. Da 144331 Elementknoten existieren, muss dazu noch die gleiche Anzahl Bytes für das Längen-Byte addiert werden. Der Speicherplatzverbrauch im Elementindex liegt folglich bei 936 Kilobytes.

*Dokumentcontainer* und Elementindex verbrauchen letztendlich bei der unkomprimierten Version zusammen 2953 Kilobytes.

Bei Verwendung der Präfix-Komprimierung sinkt der Speicherplatzverbrauch im *Dokumentcontainer* auf 1.17 Bytes pro DeweyID bzw. 321 Kilobytes für alle DeweyIDs zusammen. Addiert um das Längen-Byte ergeben sich in der Summe 596 Kilobytes.

Im Elementindex sinkt durch die Präfix-Komprimierung der berechnete Speicherplatzverbrauch von 795 Kilobytes auf 254 Kilobytes. Addiert um 144331 Bytes für das Längen-Byte ergibt die Summe für den Elementindex 395 Kilobytes.

*Dokumentcontainer* und Elementindex verbrauchen letztendlich in der komprimierten Version 991 Kilobytes. Nicht berücksichtigt ist hierbei der Speicherplatz des B-Baums, der die Liste indexiert.

Folglich müsste die Speicherung des Dokuments mit Präfix-Komprimierung knapp zwei Megabytes einsparen. Tatsächlich werden sogar 0.5 Megabytes mehr eingespart, die auf die B-Bäume und andere Indizes zurückzuführen sind.

**Tabelle 4.5** Speicherplatzverbrauch des Testdokuments

| | Indexmanager 1 | Indexmanager 2 |
|---|---|---|
| ohne Präfixkomprimierung | 29.71 MB | 25.77 MB |
| mit Präfixkomprimierung | - | 23.29 MB |

**Tabelle 4.6** Rechnerische Speicherplatzverbrauch

| | ohne Präfixkompr. | mit Präfixkompr. | Einsparung |
|---|---|---|---|
| Dokumentindex | 2017 kBytes | 596 kBytes | 1421 kBytes |
| Elementindex | 936 kBytes | 395 kBytes | 541 kBytes |
| Summe | 2953 kBytes | 991 kBytes | 1962 kBytes |

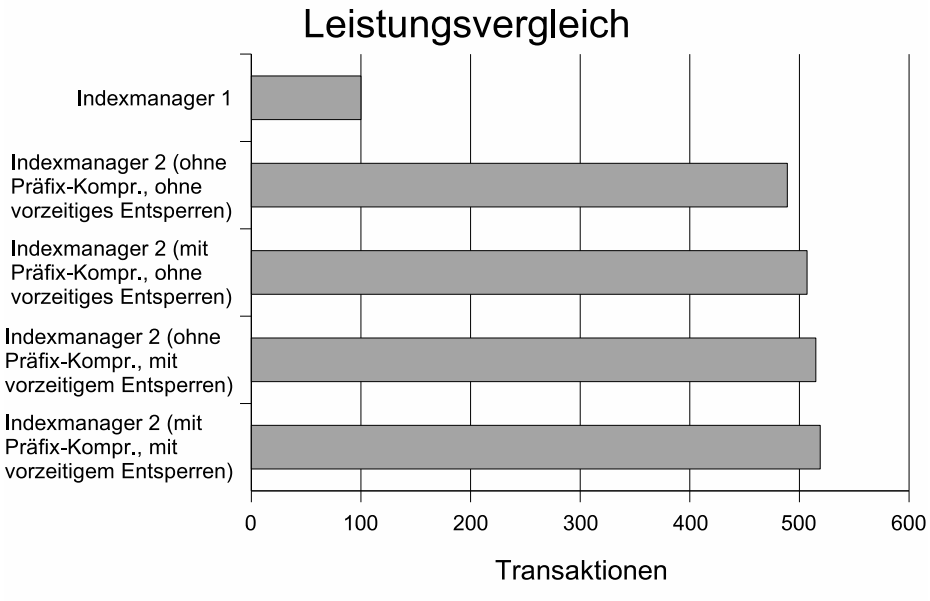
### 4.7.2 Leistung

Indexmanager 1 erreicht in diesem Testversuch 100 Transaktionen, Indexmanager 2 hingegen 489 Transaktionen, wenn keine Präfix-Komprimierung verwendet wird und auch die Seiten im B-Baum nicht vorzeitig entsperrt werden (wie in Kapitel 4.5.1 beschrieben). Bei Benutzung der Präfix-Komprimierung, jedoch ohne vorzeitiges Entsperren steigt die Transaktionszahl auf 507. Ohne Präfix-Komprimierung, aber mit vorzeitigem Entsperren sind es 515 Transaktionen. Schließlich werden mit Präfix-Komprimierung und mit vorzeitigem Entsperren 519 Transaktionen erreicht. Abbildung 4.33 stellt den Transaktionsdurchsatz in einem Diagramm dar.

An dieser Stelle kann auch gut der Einfluss getestet werden, den ein Vergleich von DeweyIDs auf Byte-Ebene (Abschnitt 3.2.4.3) auf die Transaktionszahl ausübt. Bei den oben angegebenen Zahlen findet der Vergleich der DeweyIDs auf Byte-Ebene statt. Würden dagegen bei einem Vergleich von zwei DeweyIDs erst beide DeweyID-Objekte erstellt und danach der Vergleich auf Division-Ebene durchgeführt werden, so würde beim letzten beschriebenen

Test, d. h. mit Präfix-Komprimierung und mit vorzeitigem Entsperren, der Transaktionsdurchsatz auf 502 Transaktionen sinken.

**Abbildung 4.33** Leistungsvergleich



# Kapitel 5 Zusammenfassung

Ausgangspunkt der vorliegenden Diplomarbeit ist der lauffähige XTC-Server, dessen Implementierung vor allem im Bereich der Indexstrukturen viel Verbesserungspotential besitzt. Es wird zu Beginn das taDOM-Transaktionsmodell erläutert, das im XTC die Grundlage zur Speicherung von Dokumenten und deren transaktionaler Verarbeitung bildet.

Die taDOM-Knoten werden in Datensätzen kodiert und zusammen mit DeweyIDs in Dokumentenordnung gespeichert. Die Kodierung der Datensätze kann dabei auf vier verschiedene Arten erfolgen, so dass die Möglichkeit besteht, immer die individuell effektivste Variante zu wählen: Element- und Attributnamen werden mit einem Vokabulareintrag verknüpft (*vocabulary)*, wodurch viel Speicherplatz eingespart wird, da statt des vollständigen Namens nur zwei Bytes für den Vokabularverweis verwendet werden müssen. Große Datensätze werden gespeichert, indem sie auf mehrere Seiten verteilt werden (*distributed).* Des Weiteren sind Verweise zu anderen Knoten möglich (*linked*), die in Ergebnisdokumenten verwendet werden können, um auf real existierende Knoten zu verweisen und dadurch Änderungen am Ergebnisdokument auf das Datenbanksystem propagieren zu können. Letztendlich können Datensätze materialisiert abgelegt werden (*inlined*).

DeweyIDs spielen eine Schlüsselrolle, da sie die Ebene des Knotens speichern und die Einordnung im Dokument beschreiben. Außerdem bilden sie die Grundlage zur knotenorientierten Sperrverwaltung. Sie eignen sich dafür besonders gut, da die Elternknoten ohne Dokumentenzugriff bestimmt werden können. Weiterhin können mit DeweyIDs Knotenneunummerierungen weitestgehend verhindert werden, indem man gerade Divisions zur Überlaufbehandlung verwendet. Zusätzlich zu diesen vielen positiven Eigenschaften ist es möglich, den Speicherplatzverbrauch im *Dokumentcontainer* auf 2 bis 2,5 Bytes (inklusive Längenbyte) zu beschränken. Der erste Schritt dazu ist die Implementierung der Präfix-Komprimierung, die vor allem im *Dokumentcontainer* hervorragende Ergebnisse erzielt. Durch die Anpassung der Kodierungstabelle an die Byte-Grenzen kann der Speicherplatzverbrauch nochmals weiter gesenkt werden. Aus den Kodierungstabellen, die nach der Anpassung noch zur Auswahl stehen, wird die effizienteste durch umfangreiche Auswertungen bestimmt. Auch im Elementindex führt diese Art von Kodierung zu einem akzeptablen Speicherplatzverbrauch. Einen weiteren Vorzug dieses Verfahrens, die Steigerung des Transaktionsdurchsatzes, kann man erreichen, indem man bei der Kodierung darauf achtet, dass der Vergleich von DeweyIDs auf Byte-Ebene stattfinden kann. Aufgrund der dargelegten positiven Eigenschaften eignen sich insgesamt betrachtet die DeweyIDs sehr gut zur Knotennummerierung in XML-Datenbanksystemen.

Als Indexstrukturen, die der Satzmanager zur Speicherung der Datensätze benötigt, werden B*-Bäume eingesetzt. Deren Effizienz kann gesteigert werden, indem die Traversierungsfolge ausgehend von der Wurzelseite hin zur Blattseite gespeichert wird und manche Seiten bereits vor Transaktionsende freigegeben werden, sofern dies die Speicherplatzbelegung zulässt. Dadurch ist es möglich, dass mehrere Transaktionen parallel auf einen B-Baum zugreifen.

Ob diese implementierten Speicherungsstrukturen ausreichen werden, muss sich zeigen. Beispielsweise wird die XQuery-Verarbeitung vermutlich weitere Speicherungsstrukturen verwenden, die noch implementiert werden müssen. Sollte es bei der Auswertung von Planoperatoren bei XQuery-Anfragen zu Speicherengpässen kommen, könnten die DeweyIDs im Hauptspeicher effizienter verwendet werden. Dazu werden nicht wie jetzt alle Divisions in einem Integer-Array im Objekt gespeichert, sondern es wird allein die Byte-Repräsentation gespeichert und nur bei konkreter Anfrage eine Dekodierung der Divisions vorgenommen. Das hätte aber zur Folge, dass jede Abfrage der Divisions deutlich mehr Prozessorlast erzeugt, die die Vorteile der besseren Speicherplatzausnutzung vermutlich aufheben würde, da bei der Erzeugung eines DeweyID-Objektes etwa 95 Prozent der Zeit zur Dekodierung benötigt werden. Diese Prozessorlast würde bei dieser Art der Implementierung bei jeder Abfrage der Divisions entstehen. Dem könnte man entgegenwirken, indem die Abfrage der Divisions auf ein Minimum reduziert wird. Dementsprechend müssten die Komponenten, die mit DeweyIDs arbeiten, angepasst werden, wobei darauf zu achten ist, dass nicht einfach eine Verlagerung der Divisions in andere Komponenten stattfindet.

# Anhang A

# Detaillierte Auswertung im Dokumentcontainer

**Abbildung A.1** Kodierungstabelle 1

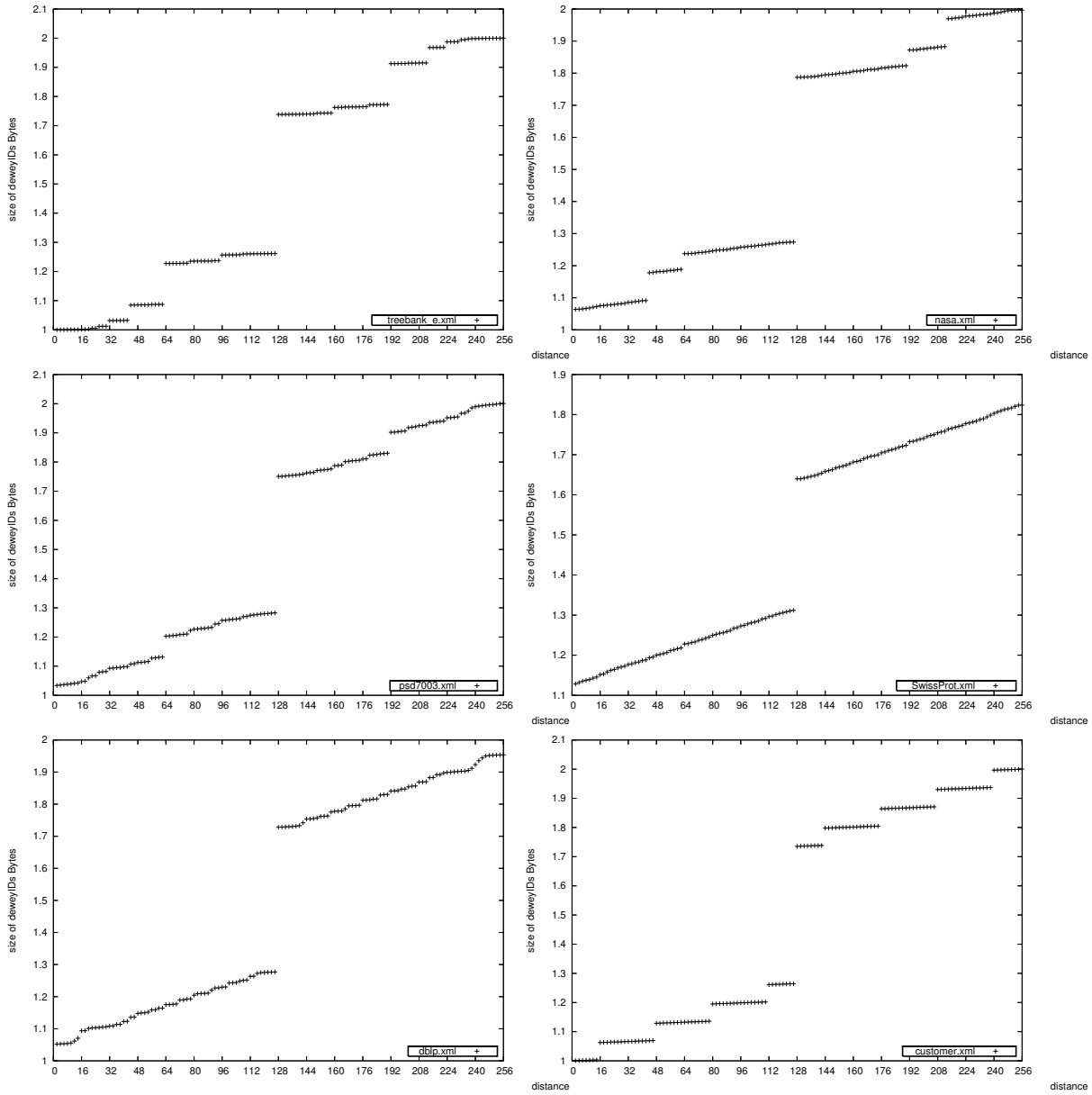


**Abbildung A.2** Kodierungstabelle 1 Fortsetzung

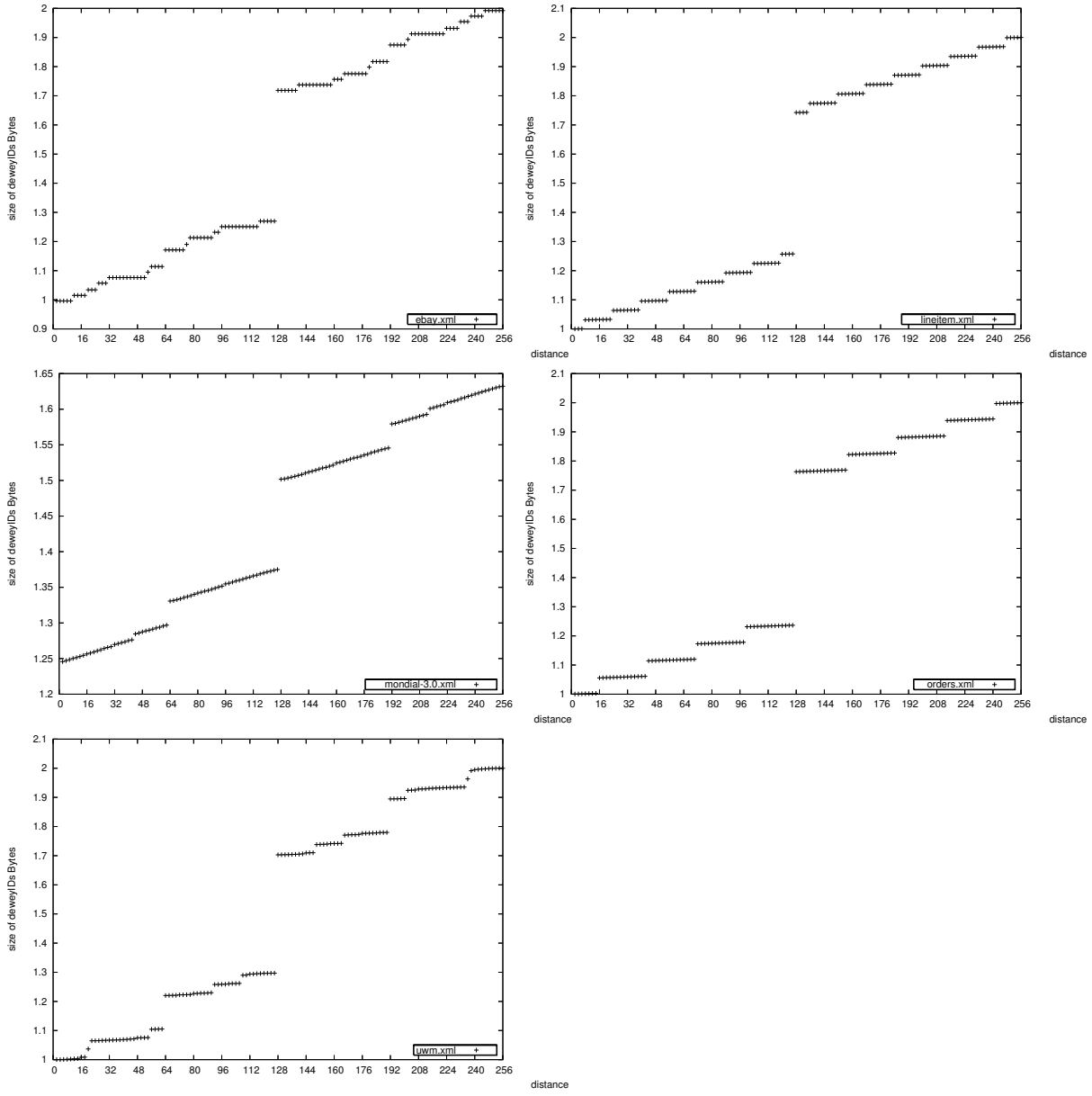



**Abbildung A.3** Kodierungstabelle 2

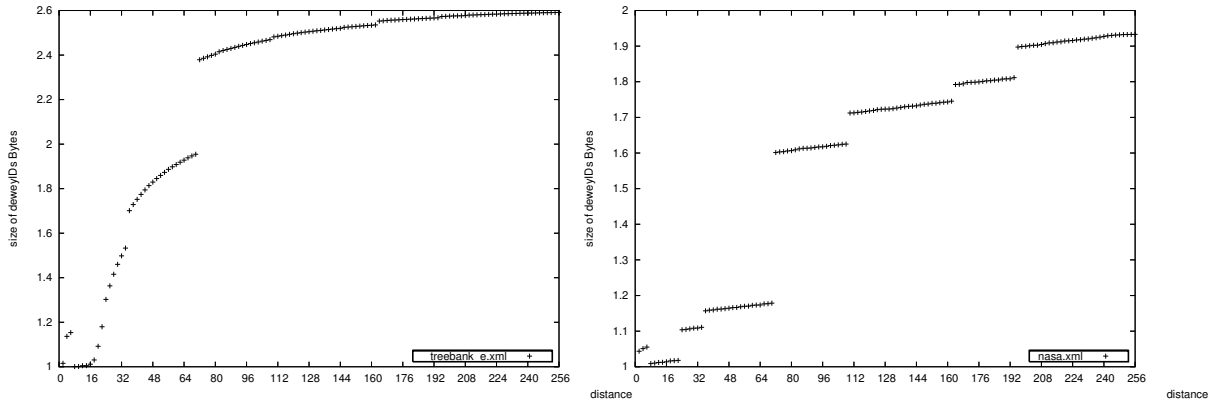


**Abbildung A.4** Kodierungstabelle 2 Fortsetzung

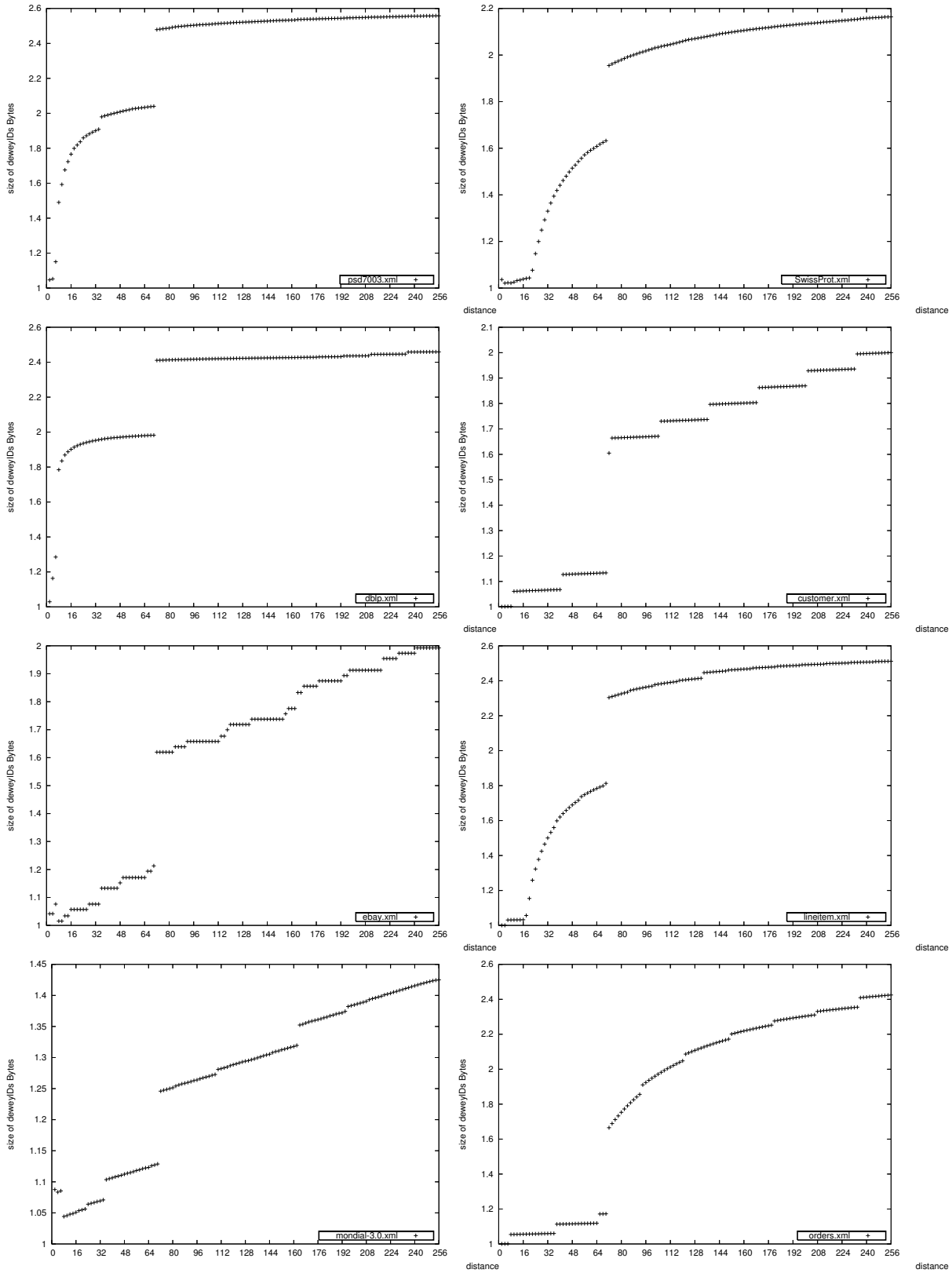


**Abbildung A.5** Kodierungstabelle 2 Fortsetzung

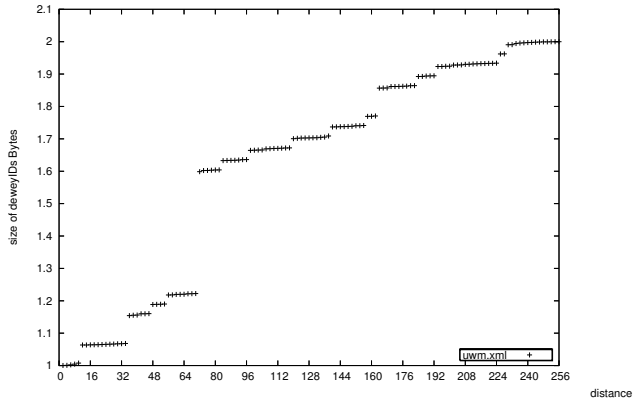



**Abbildung A.6** Kodierungstabelle 3

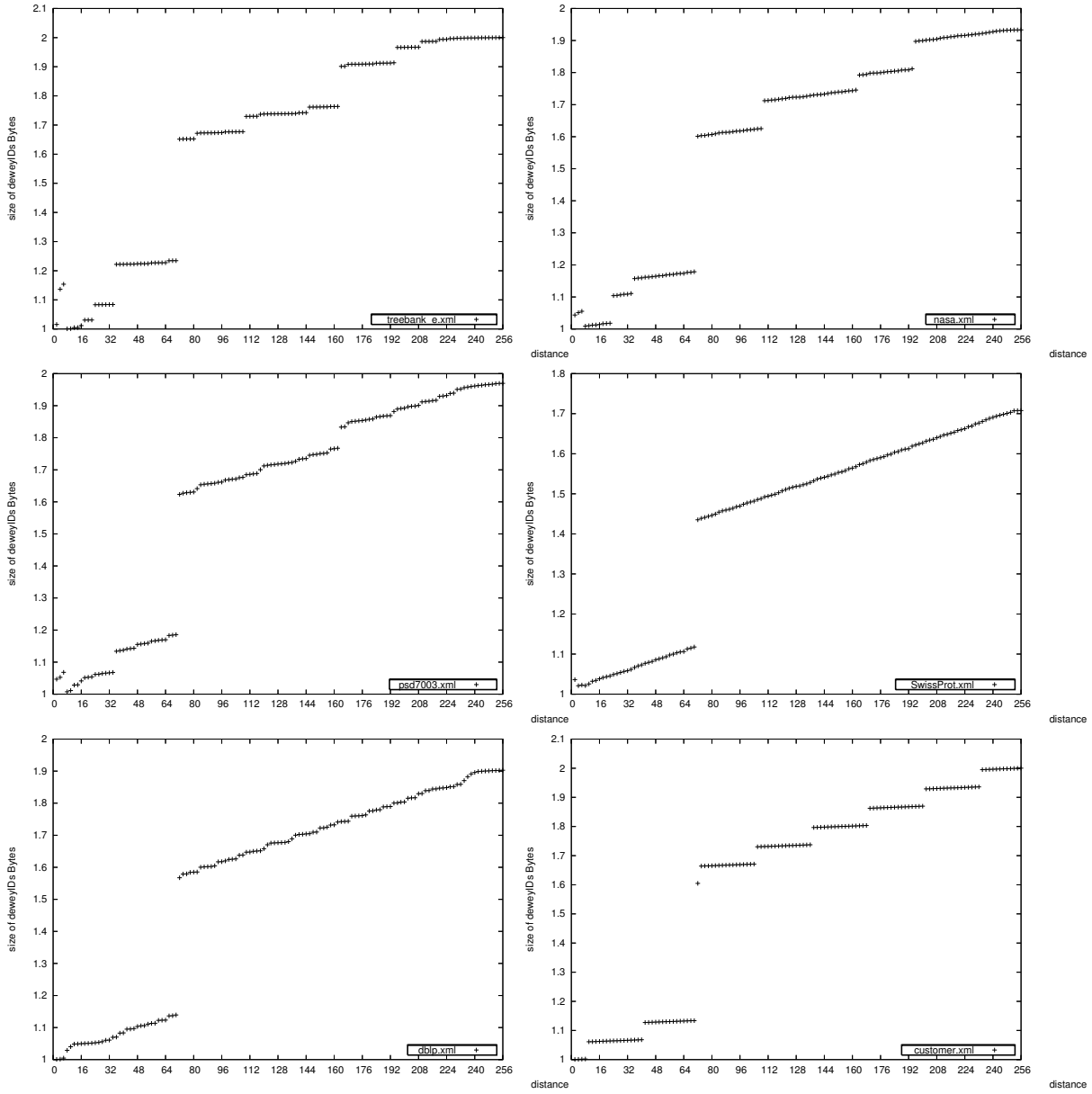


**Abbildung A.7** Kodierungstabelle 3 Fortsetzung

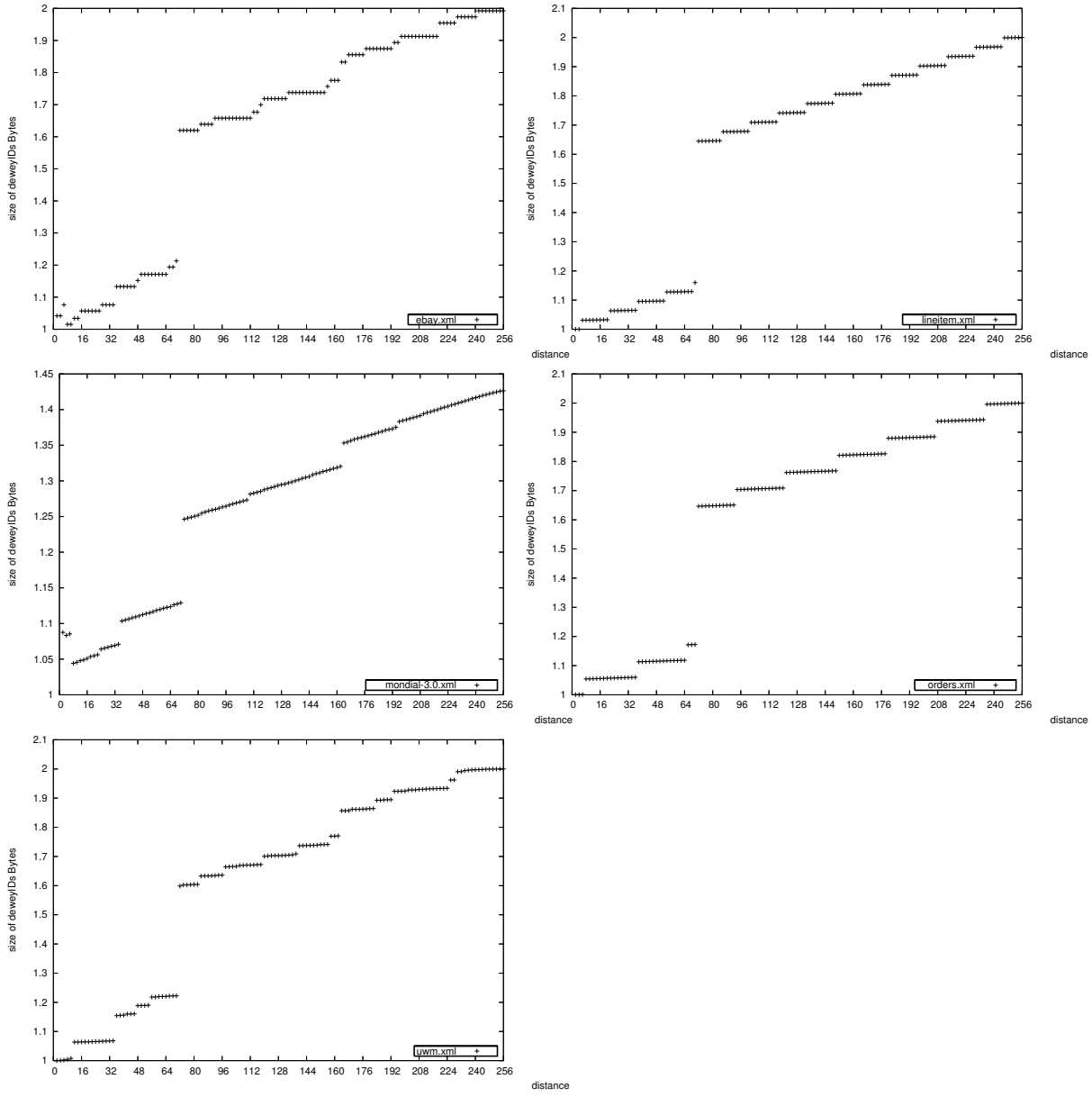



**Abbildung A.8** Kodierungstabelle 4

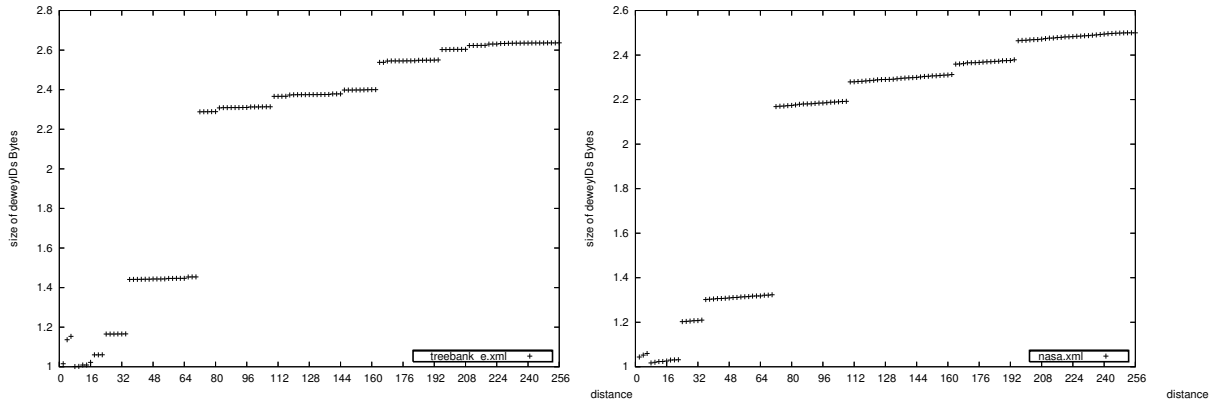


**Abbildung A.9** Kodierungstabelle 4 Fortsetzung

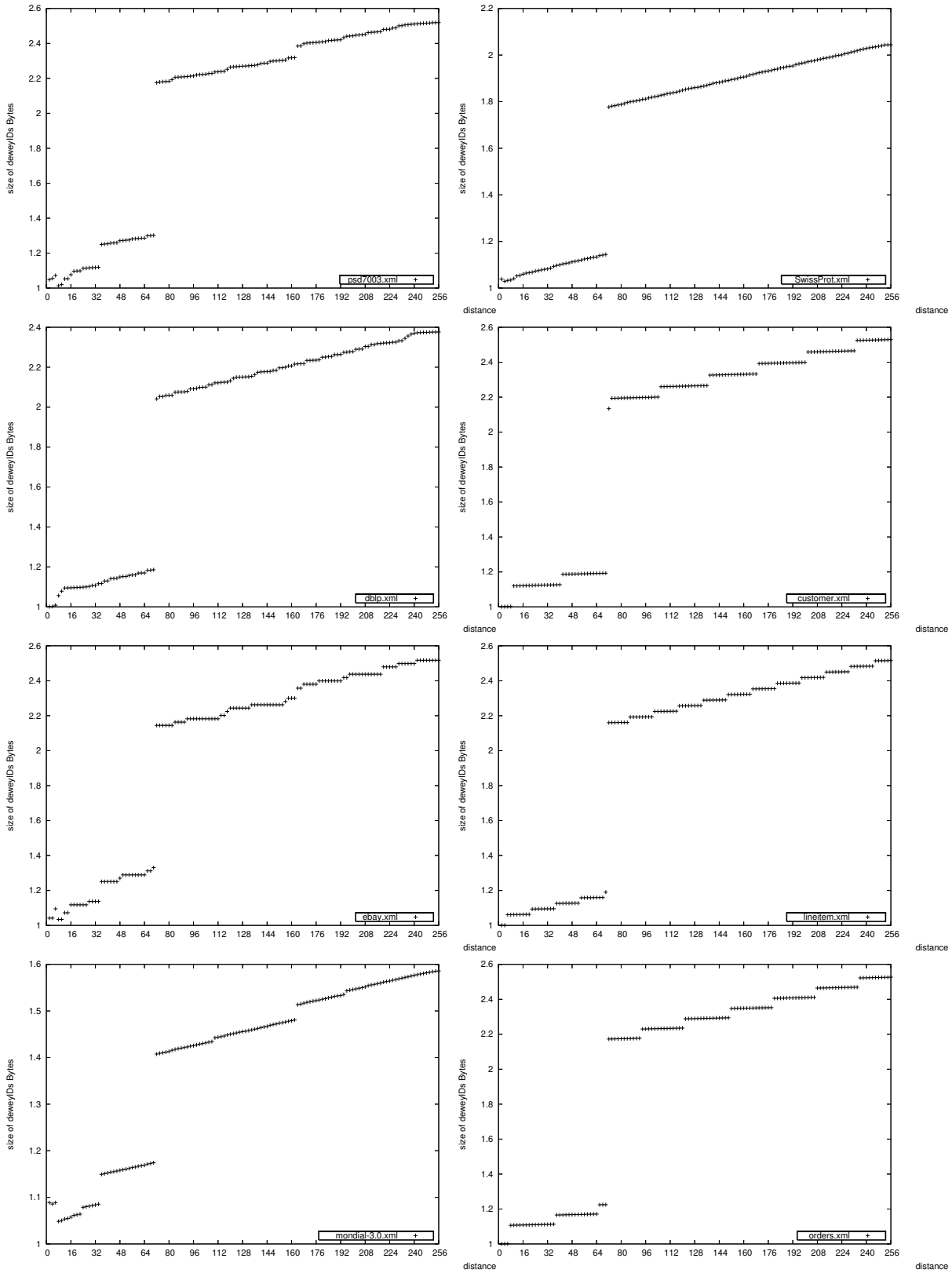


**Abbildung A.10** Kodierungstabelle 4 Fortsetzung

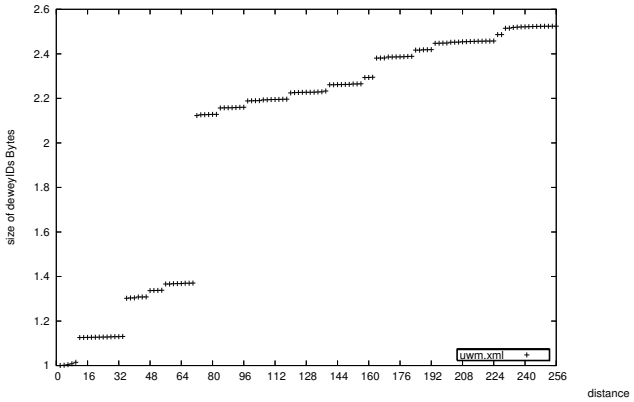



**Abbildung A.11** Kodierungstabelle 5

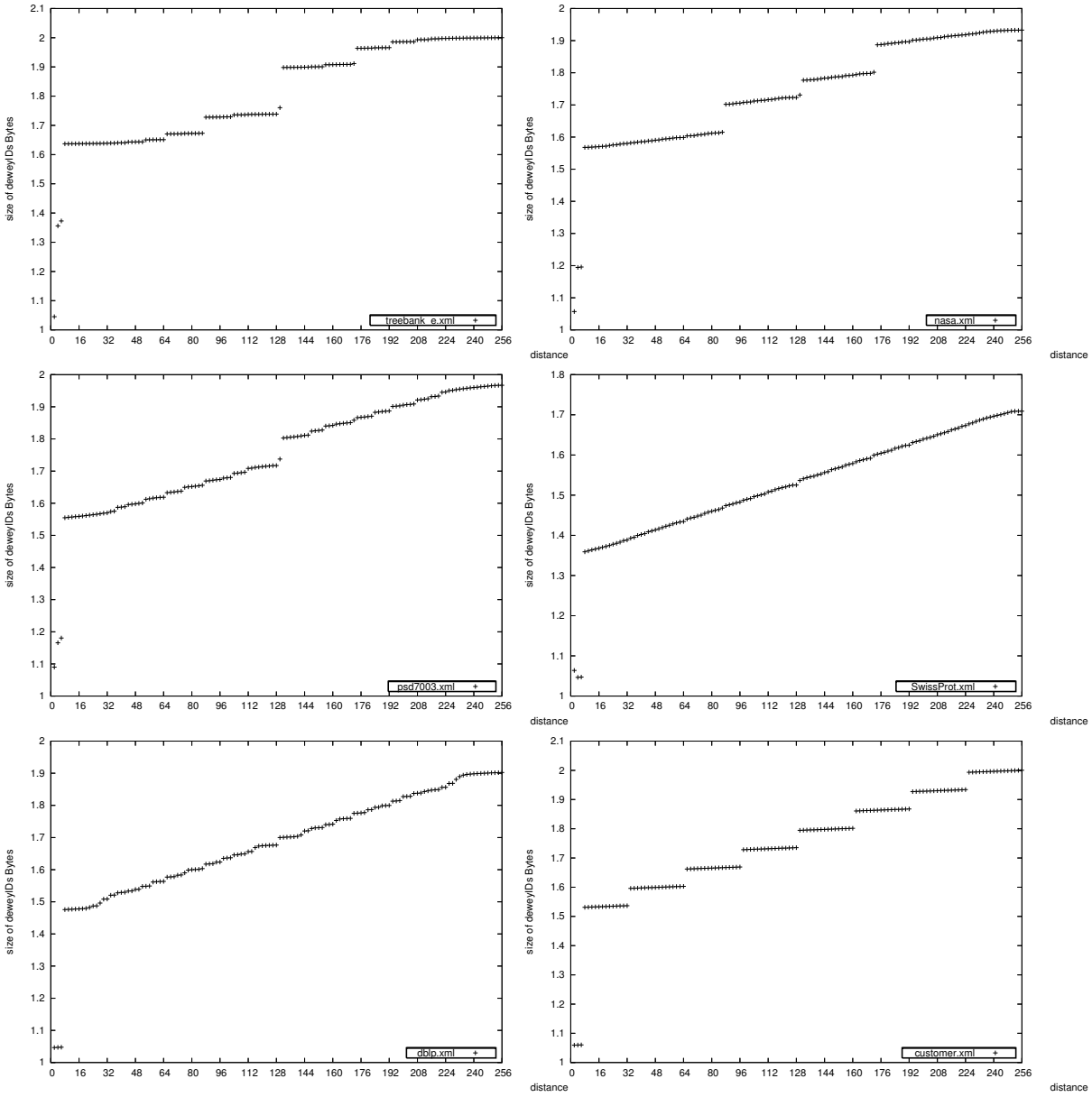


**Abbildung A.12** Kodierungstabelle 5 Fortsetzung

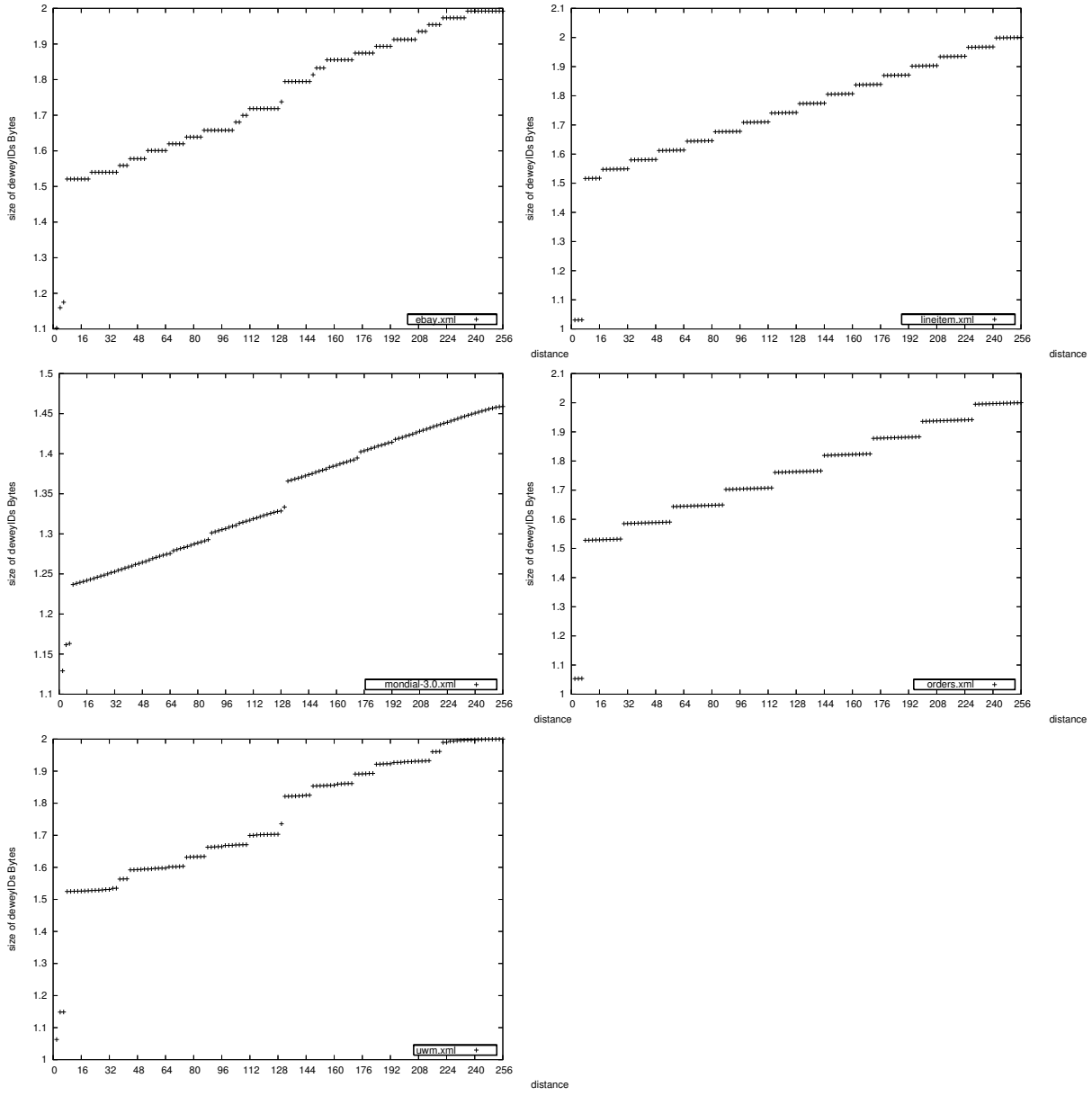



**Abbildung A.13** Kodierungstabelle 6

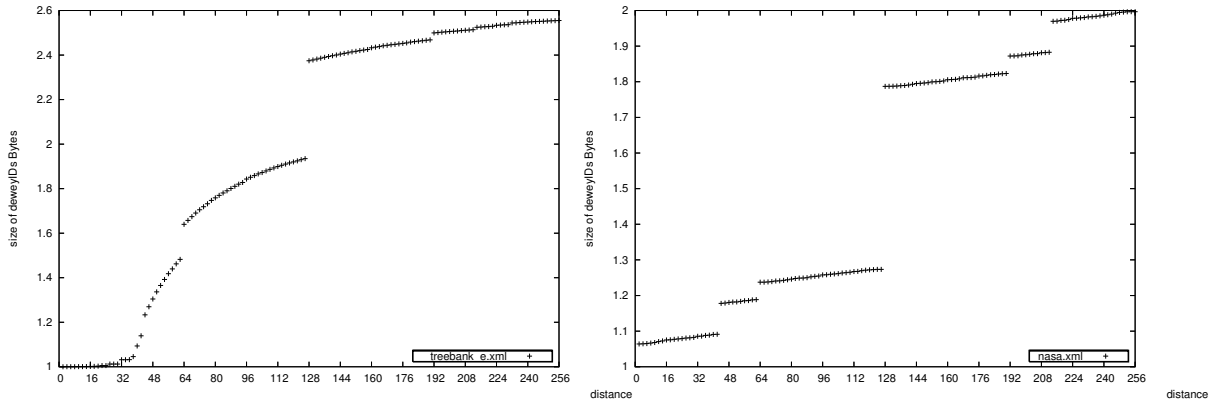


**Abbildung A.14** Kodierungstabelle 6 Fortsetzung

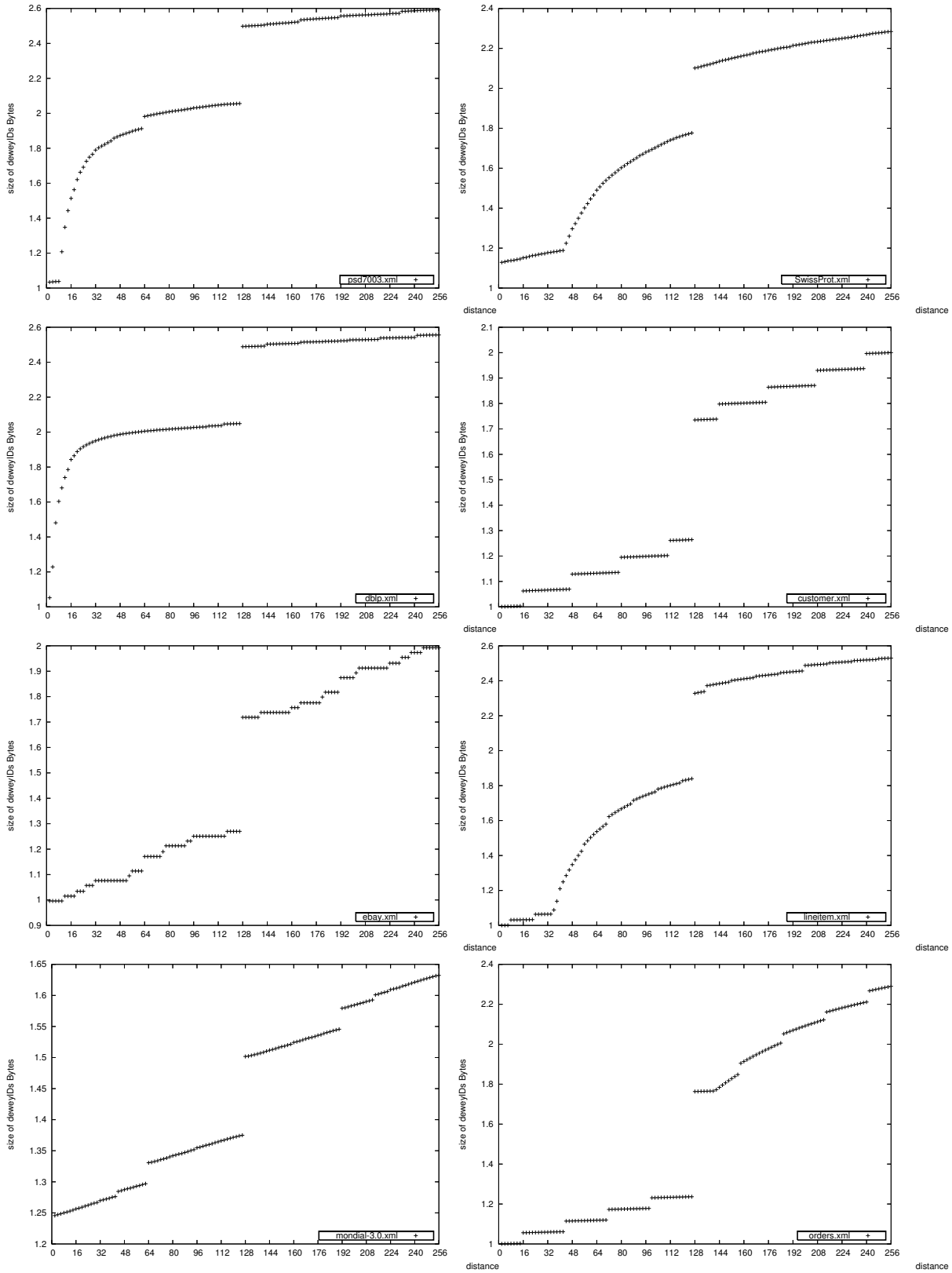


**Abbildung A.15** Kodierungstabelle 6 Fortsetzung

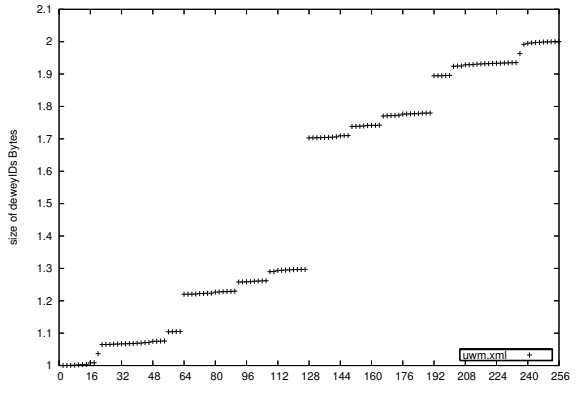



**Abbildung A.16** Kodierungstabelle 7

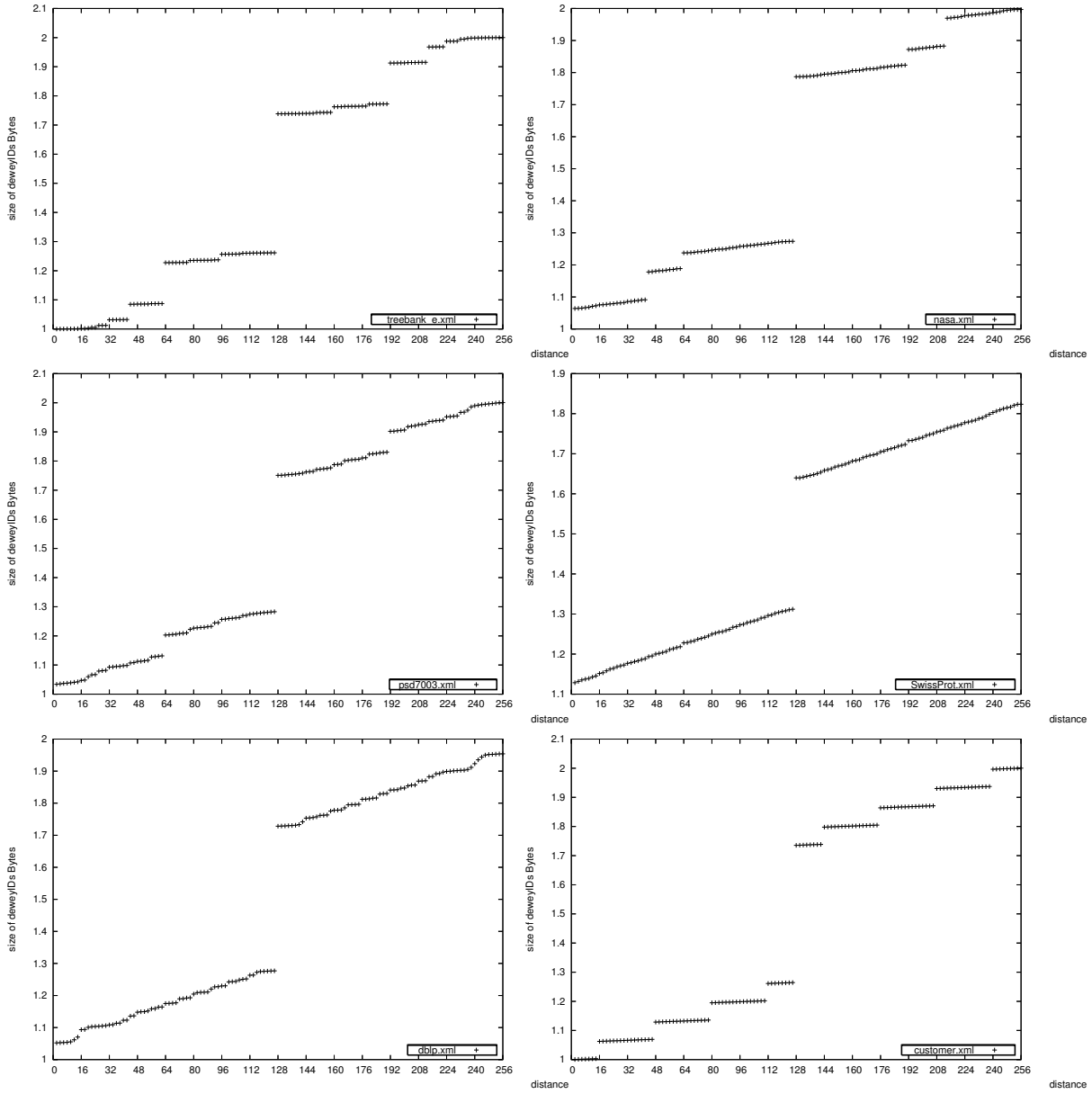


**Abbildung A.17** Kodierungstabelle 7 Fortsetzung

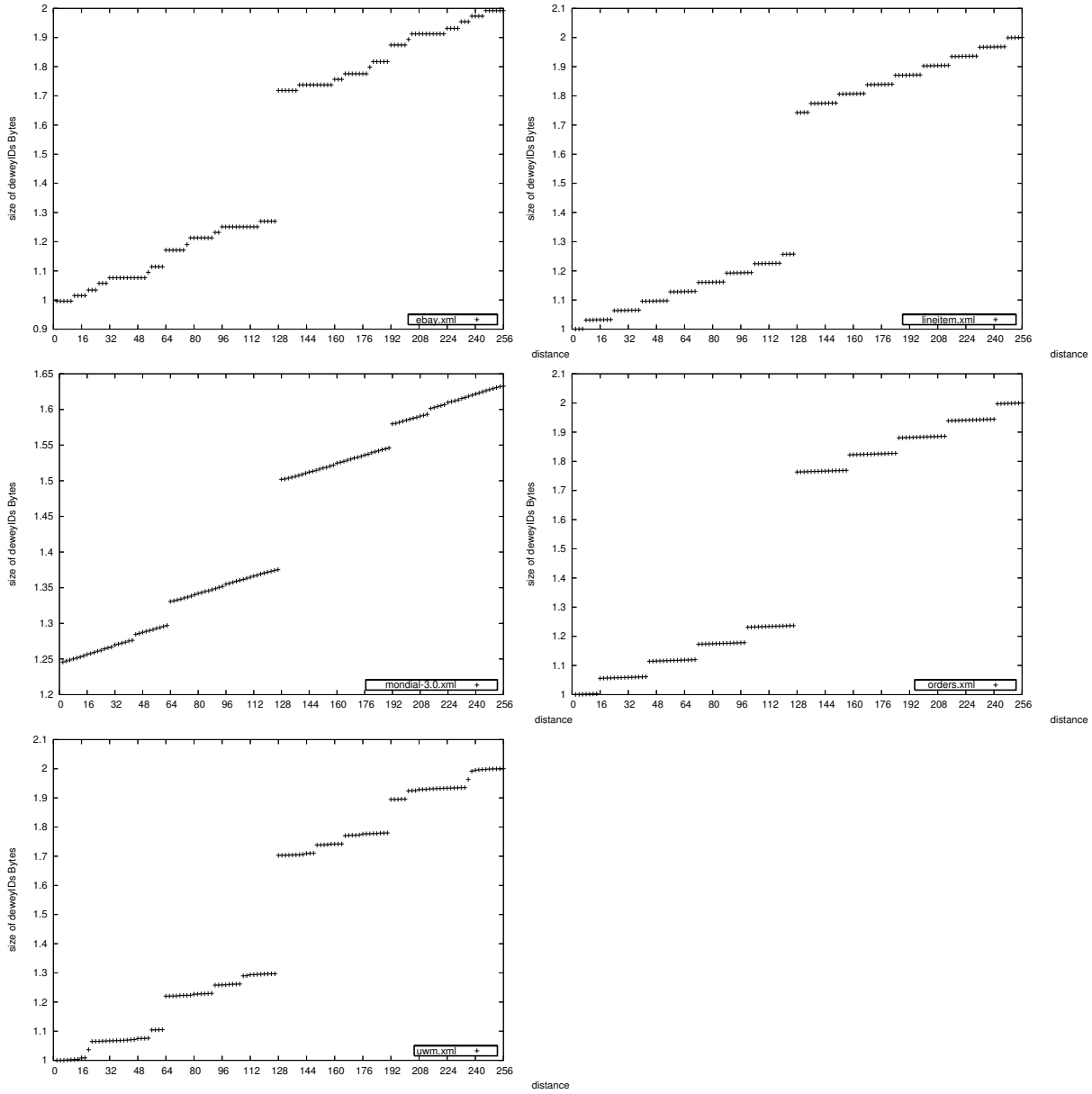



**Abbildung A.18** Kodierungstabelle 8

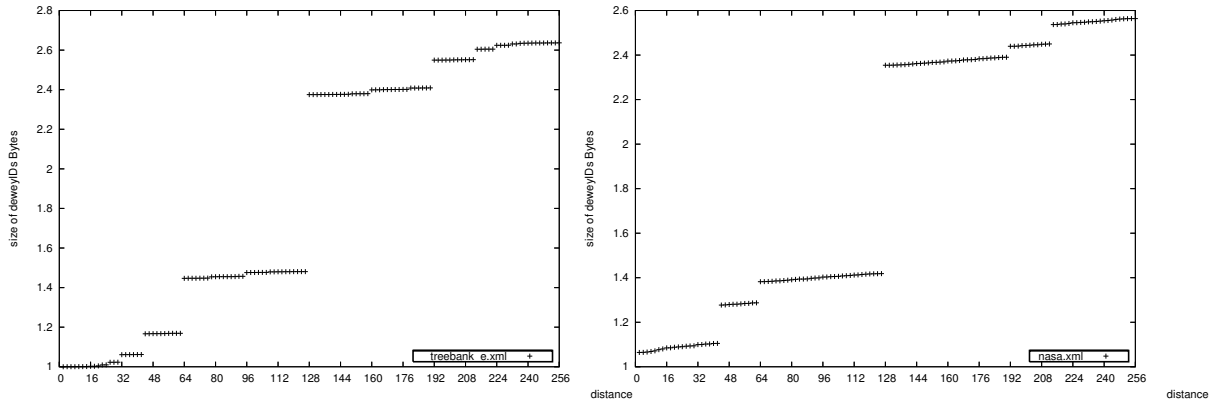


**Abbildung A.19** Kodierungstabelle 8 Fortsetzung

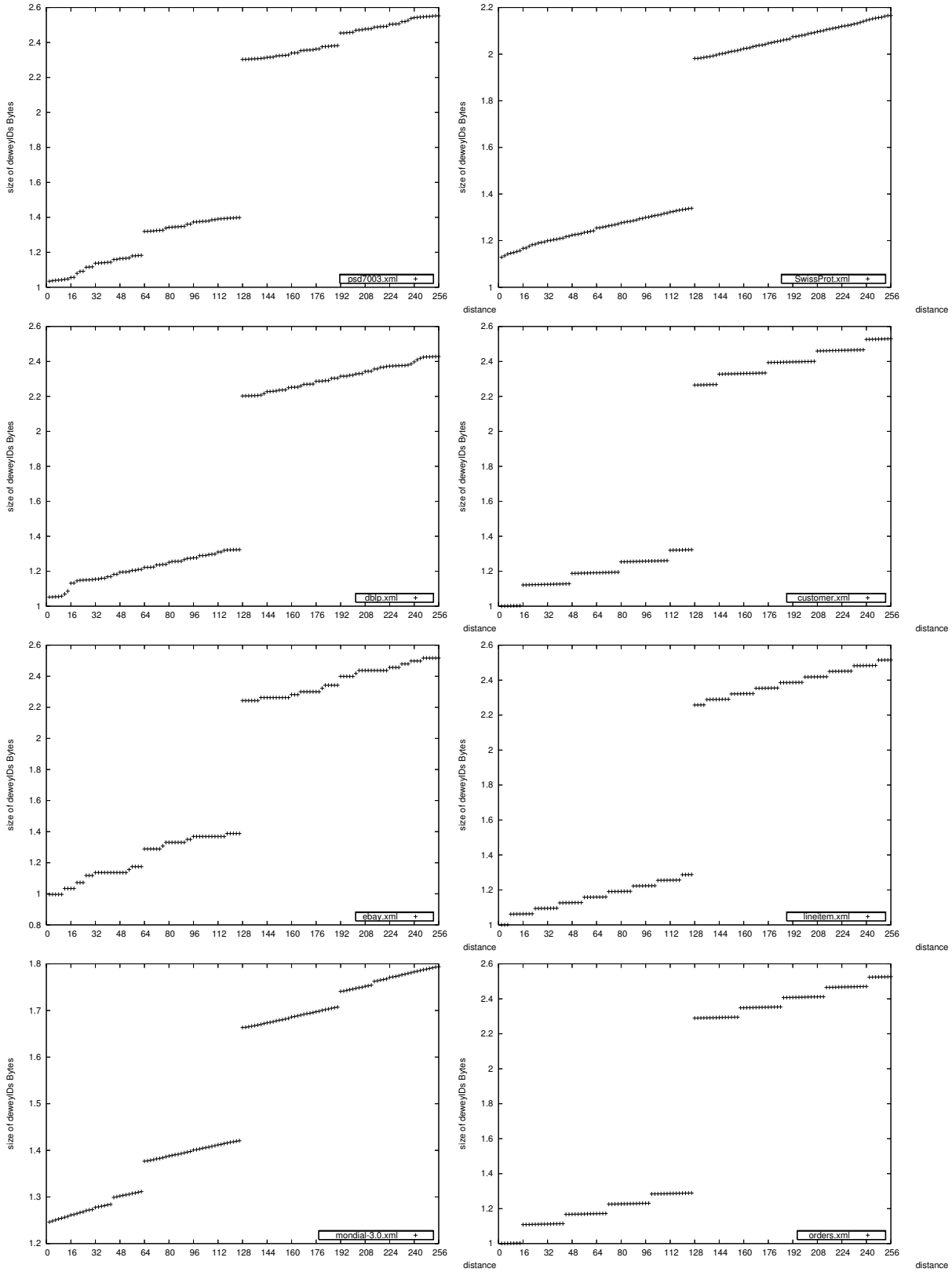


**Abbildung A.20** Kodierungstabelle 8 Fortsetzung

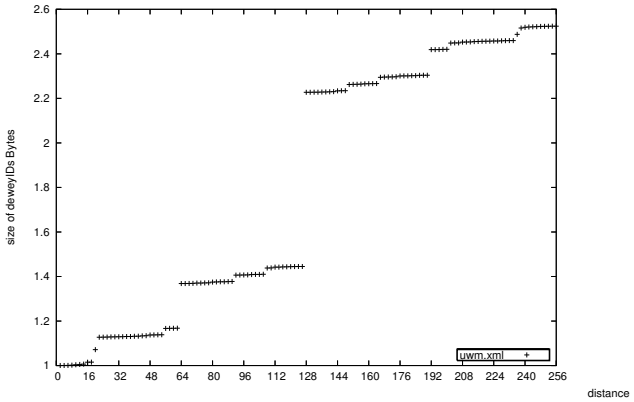



**Abbildung A.21** Kodierungstabelle 9

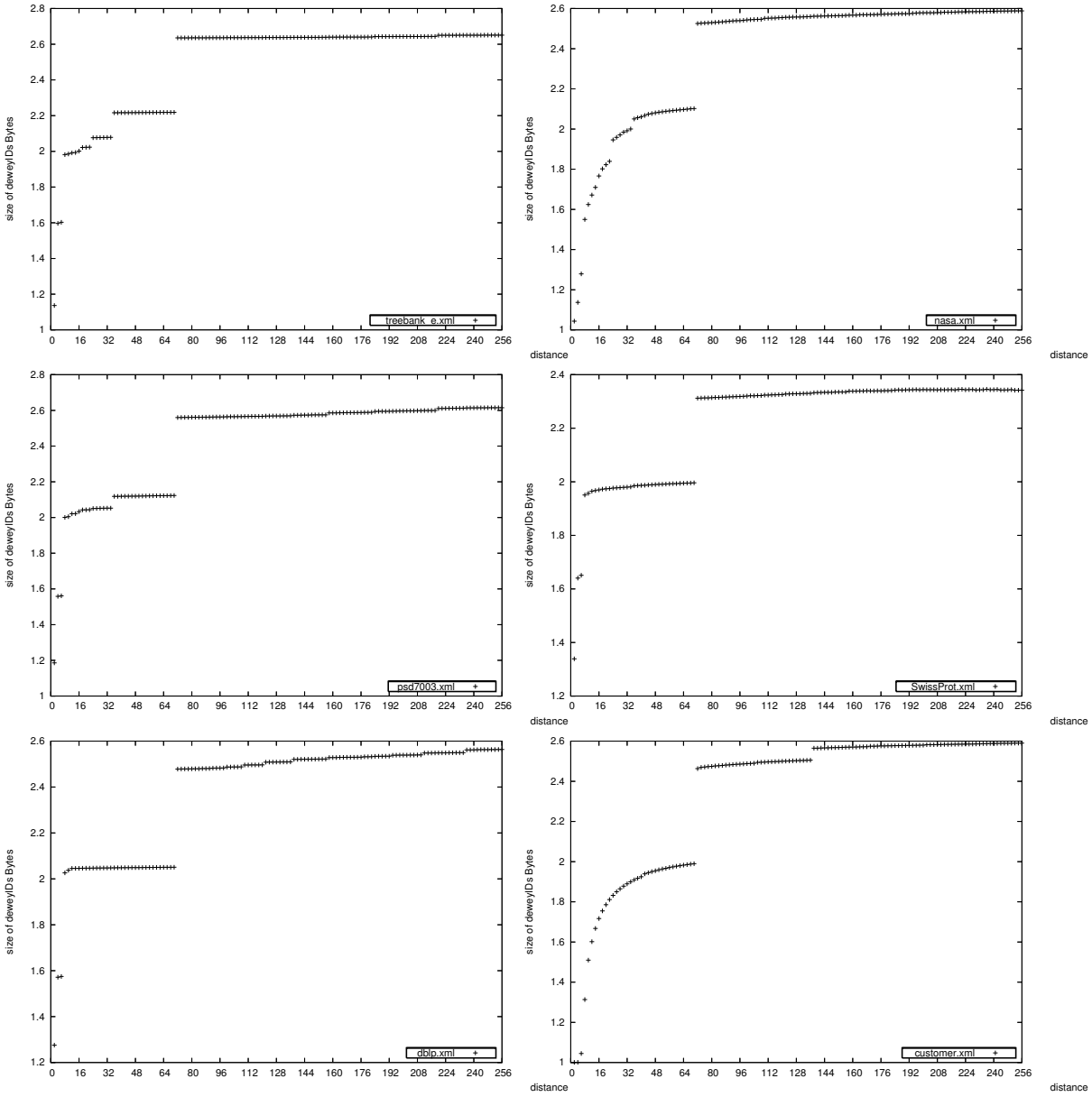


**Abbildung A.22** Kodierungstabelle 9 Fortsetzung

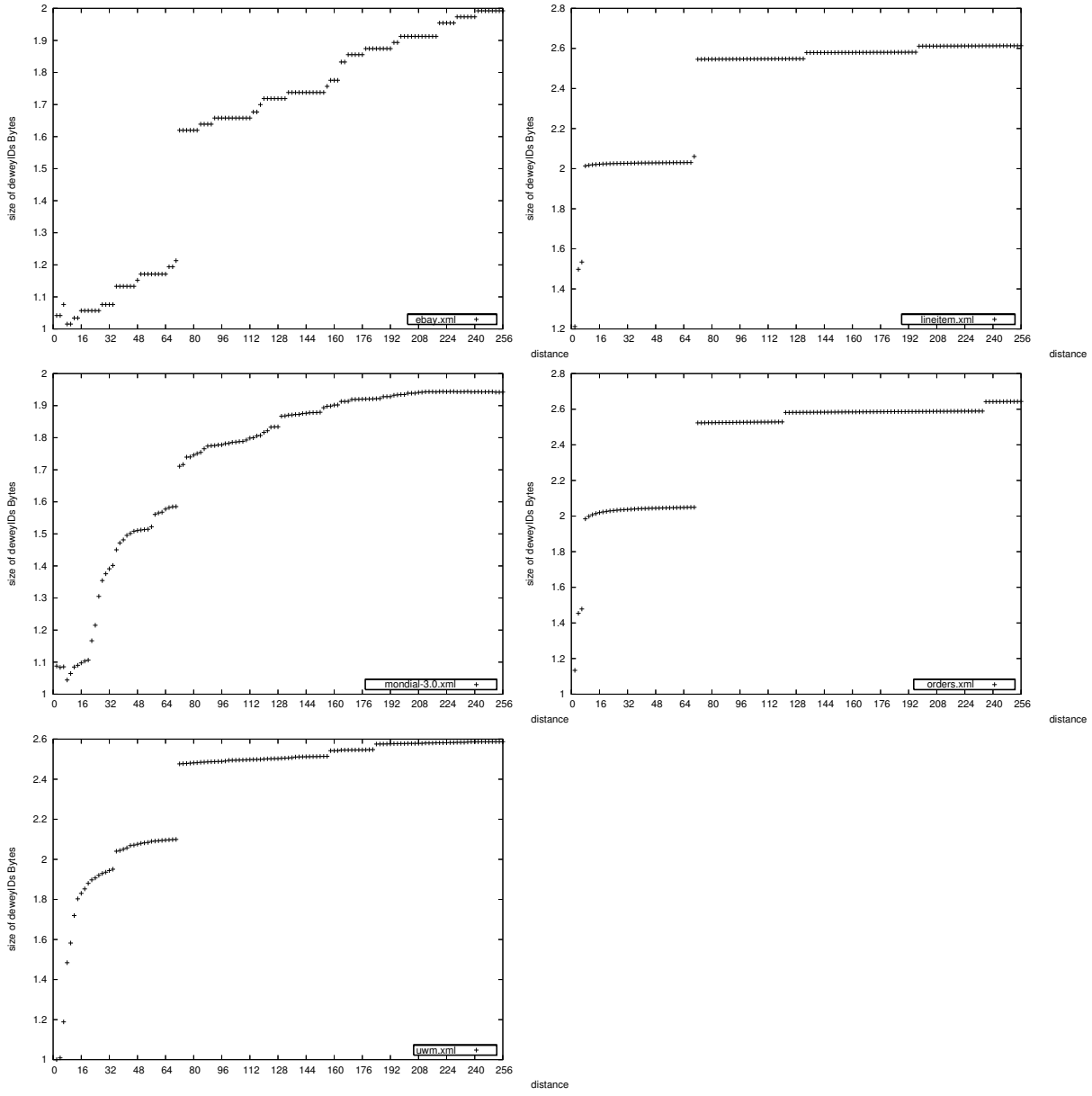



**Abbildung A.23** Kodierungstabelle 10

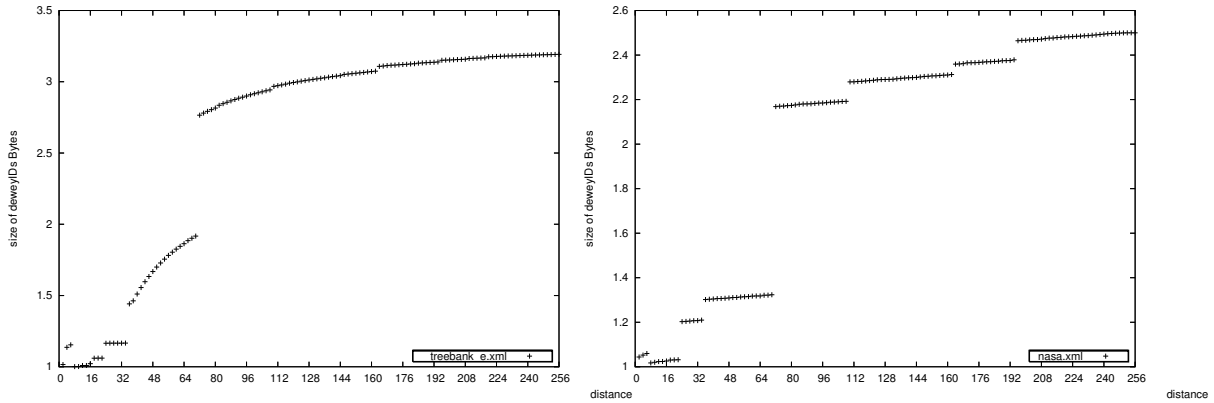


**Abbildung A.24** Kodierungstabelle 10 Fortsetzung

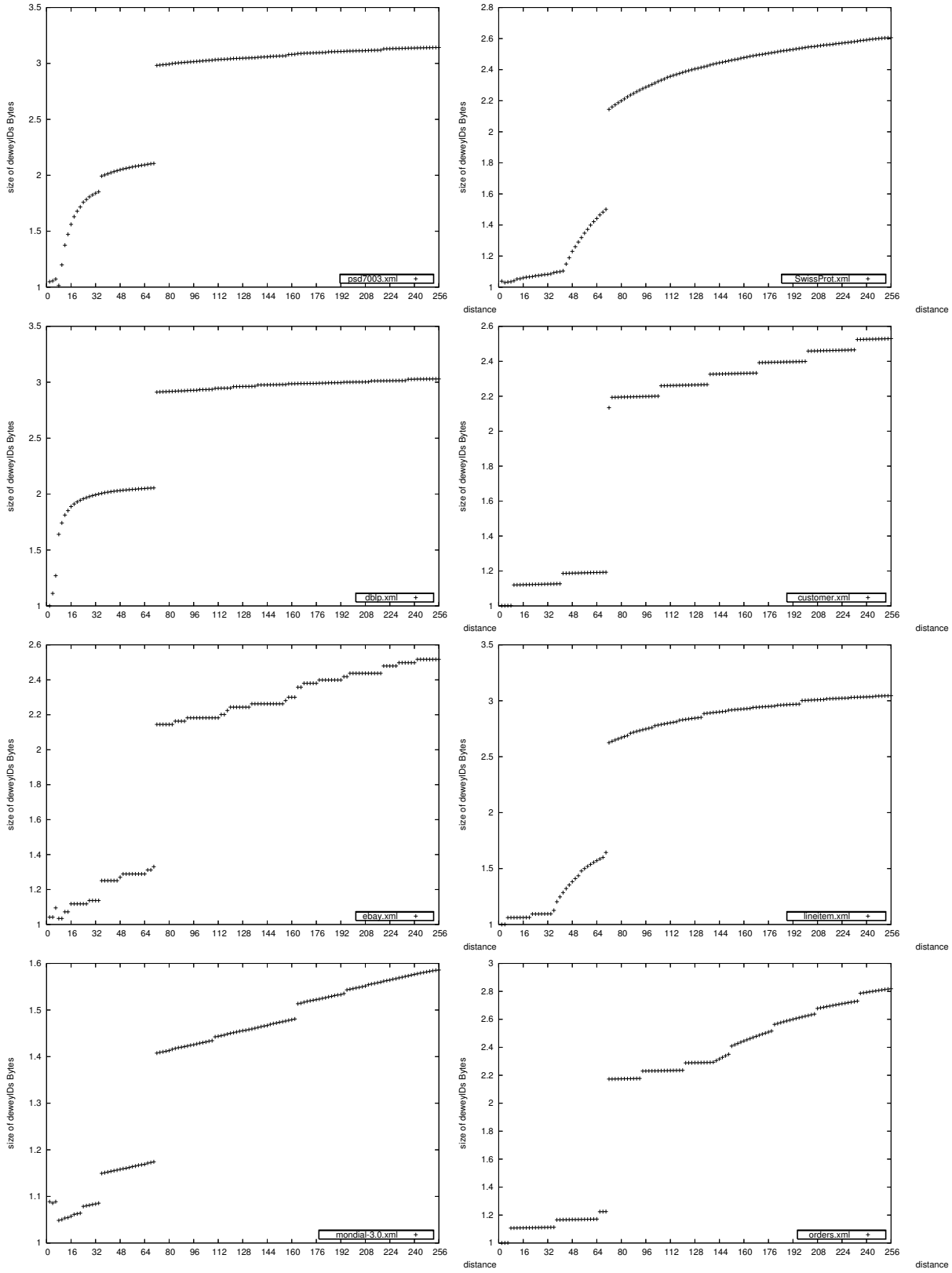


**Abbildung A.25** Kodierungstabelle 10 Fortsetzung

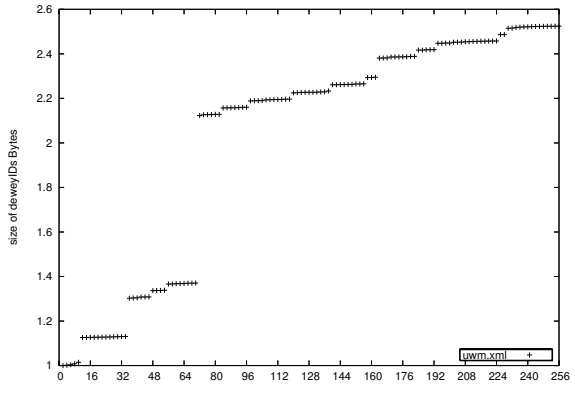



**Abbildung A.26** Kodierungstabelle 11

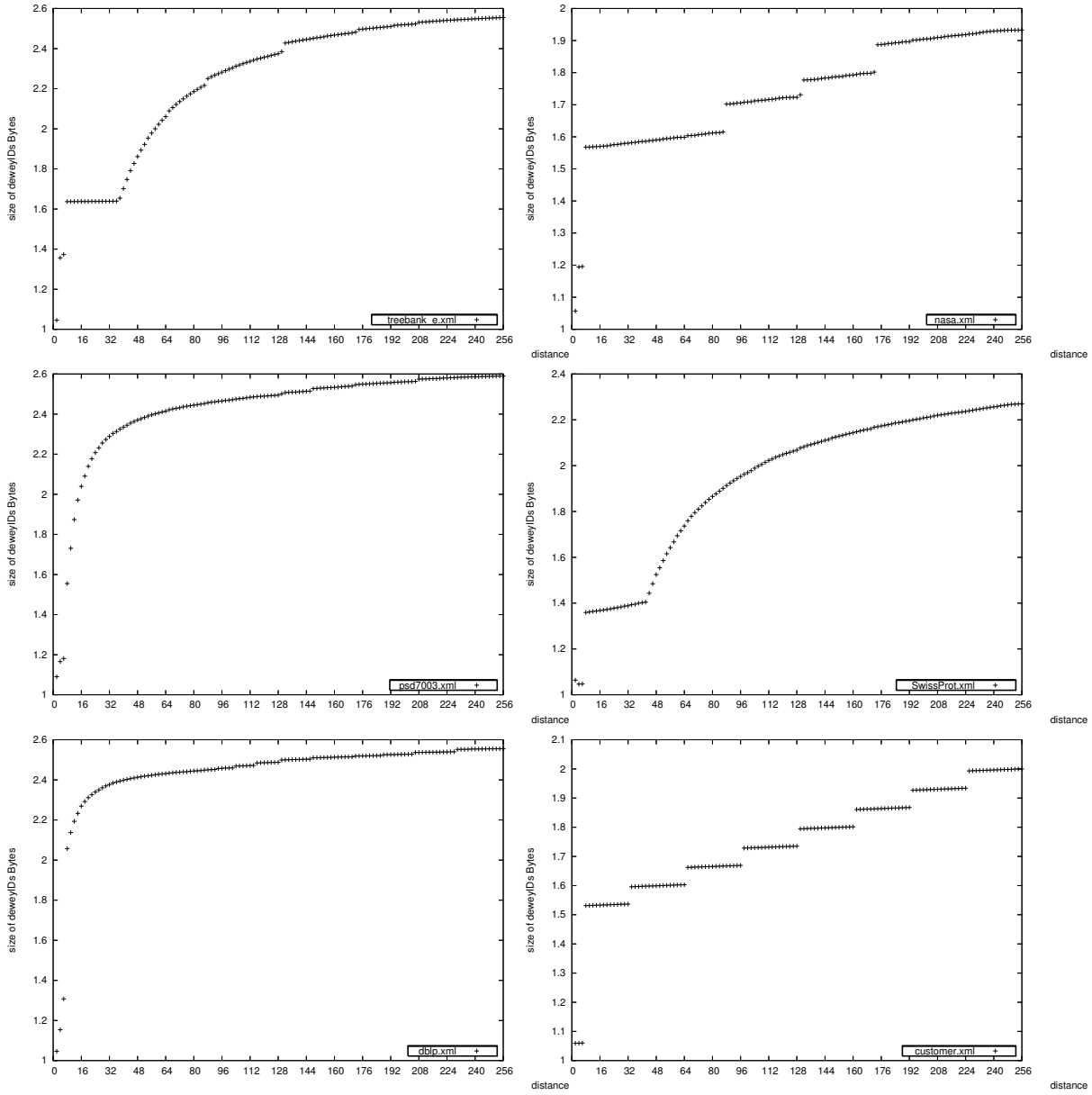


**Abbildung A.27** Kodierungstabelle 11 Fortsetzung

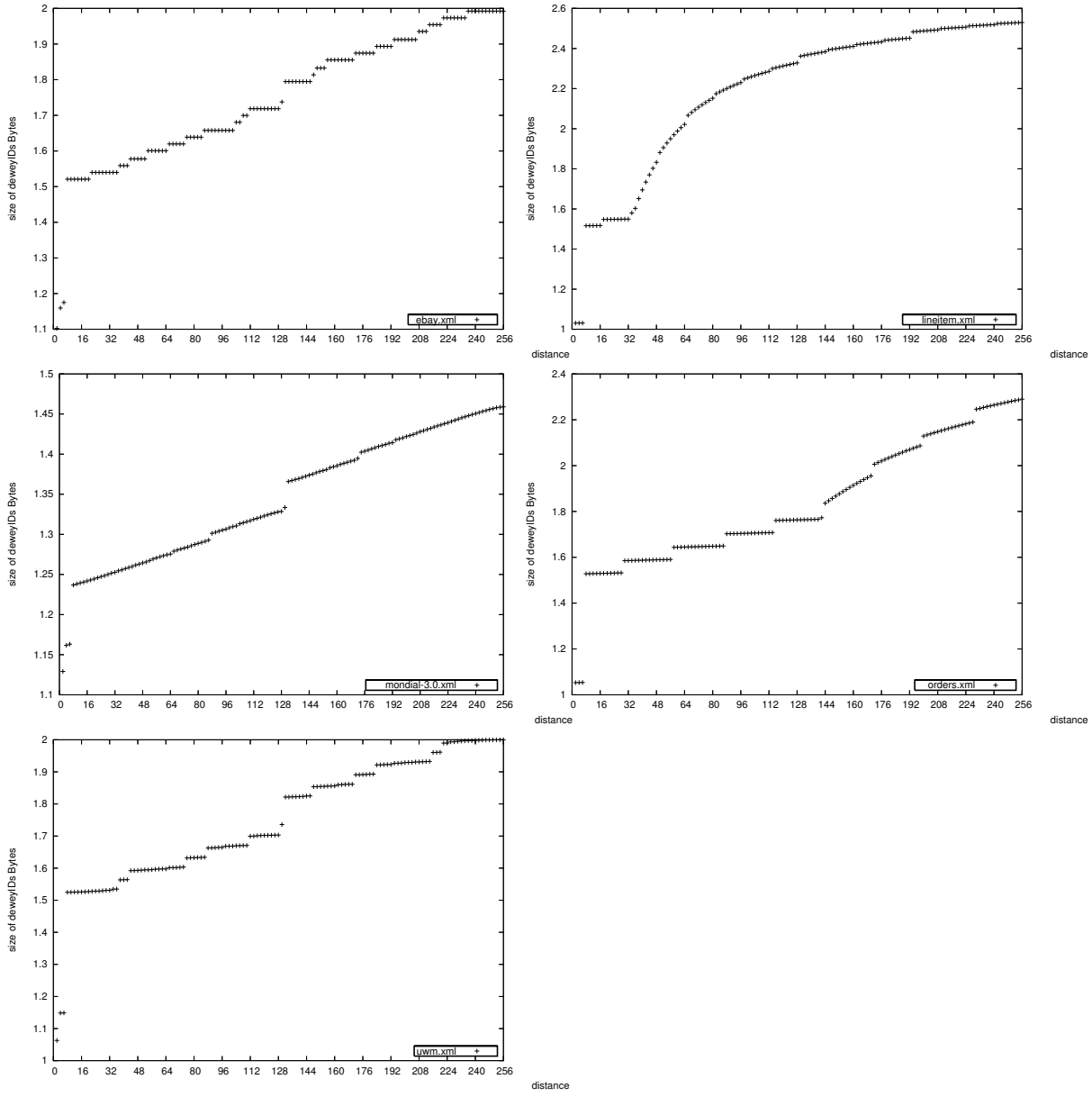



**Abbildung A.28** Kodierungstabelle 12

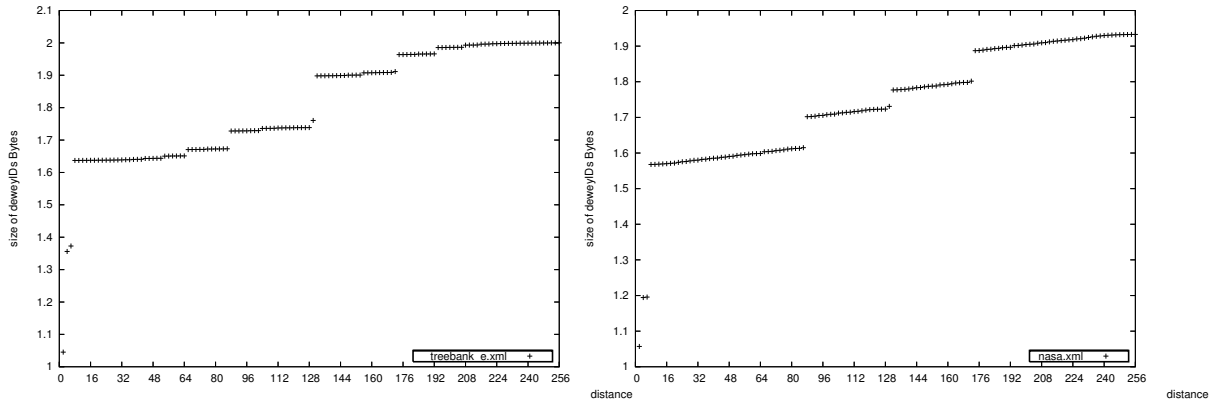


**Abbildung A.29** Kodierungstabelle 12 Fortsetzung

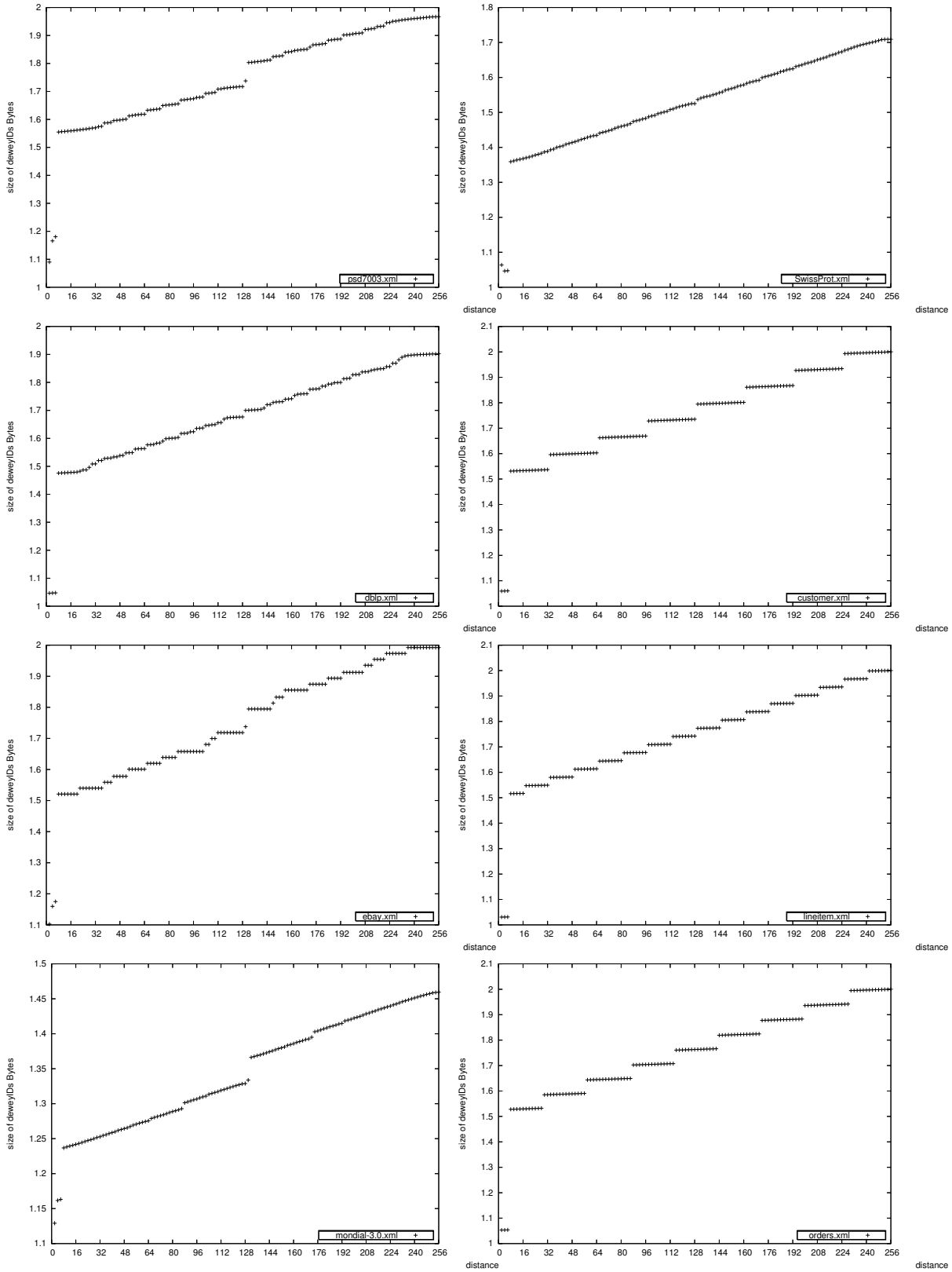


**Abbildung A.30** Kodierungstabelle 12 Fortsetzung

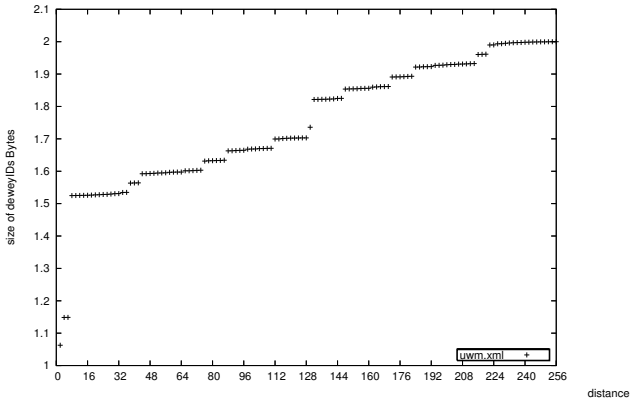



**Abbildung A.31** Kodierungstabelle 13

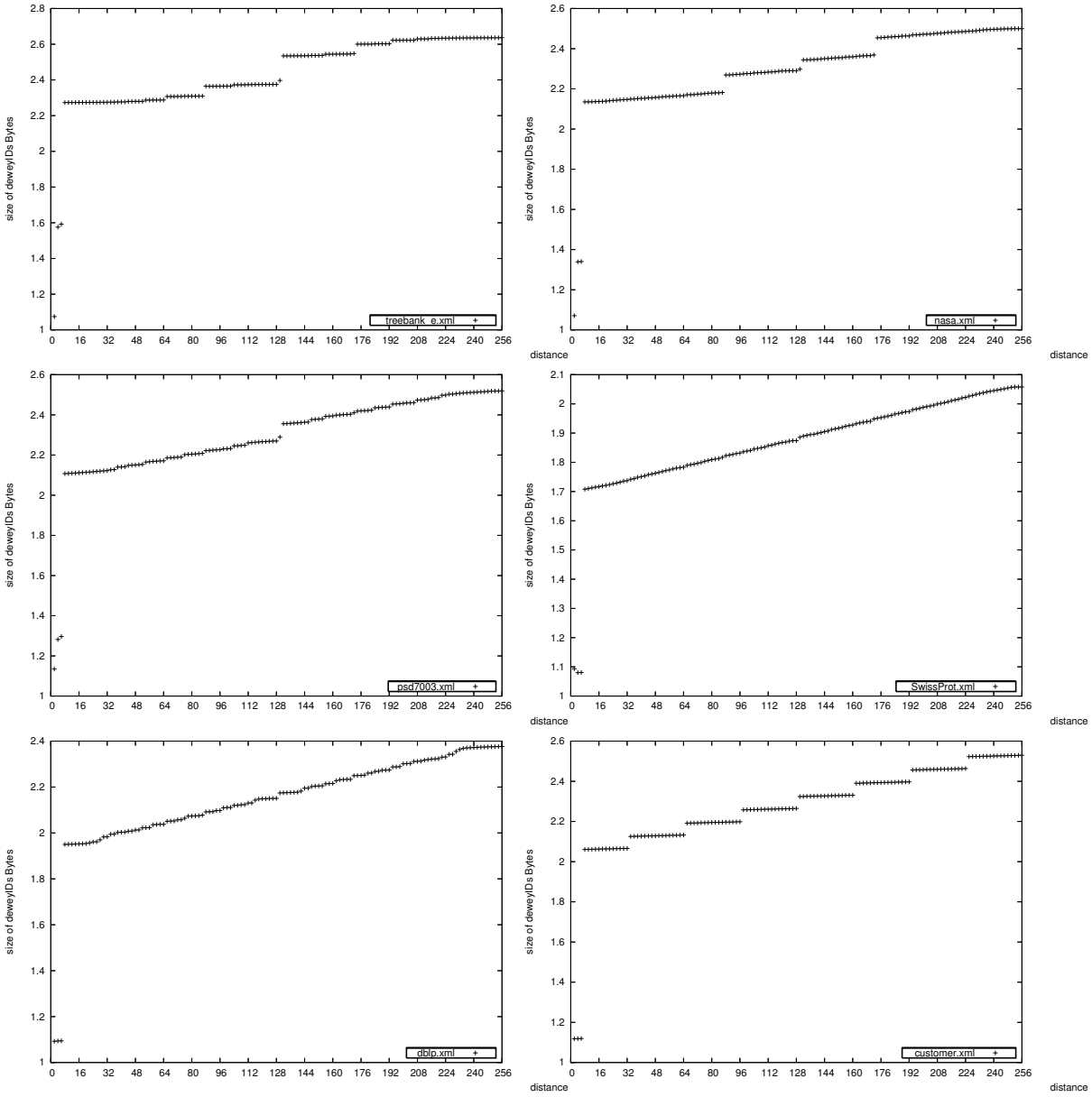
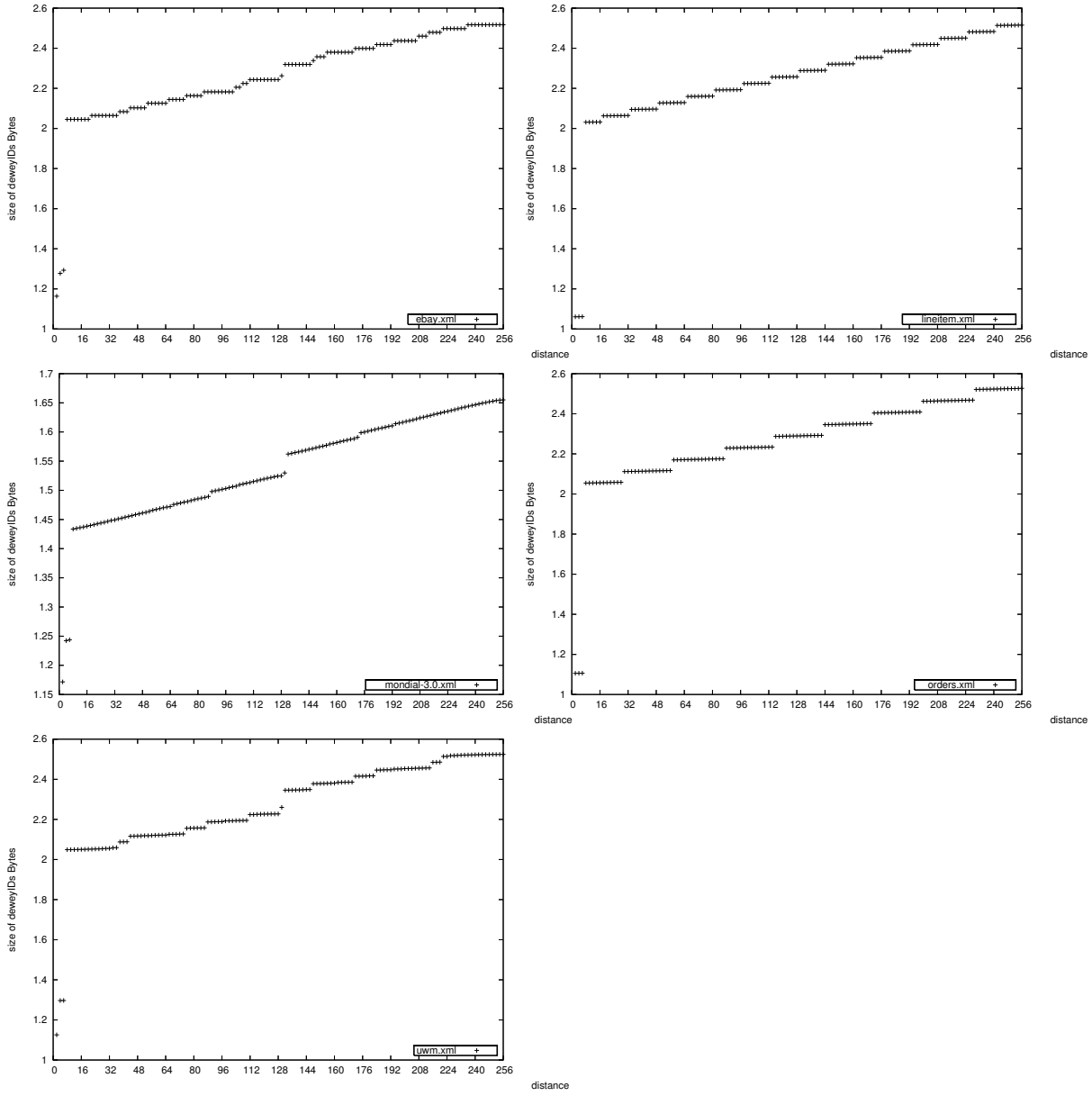


**Abbildung A.32** Kodierungstabelle 13 Fortsetzung



Anhang B

# Vergleich der maximalen DeweyID-Länge von K1 und K3

**Abbildung B.1** K1 im Vergleich zu K3: maximale DeweyID-Länge

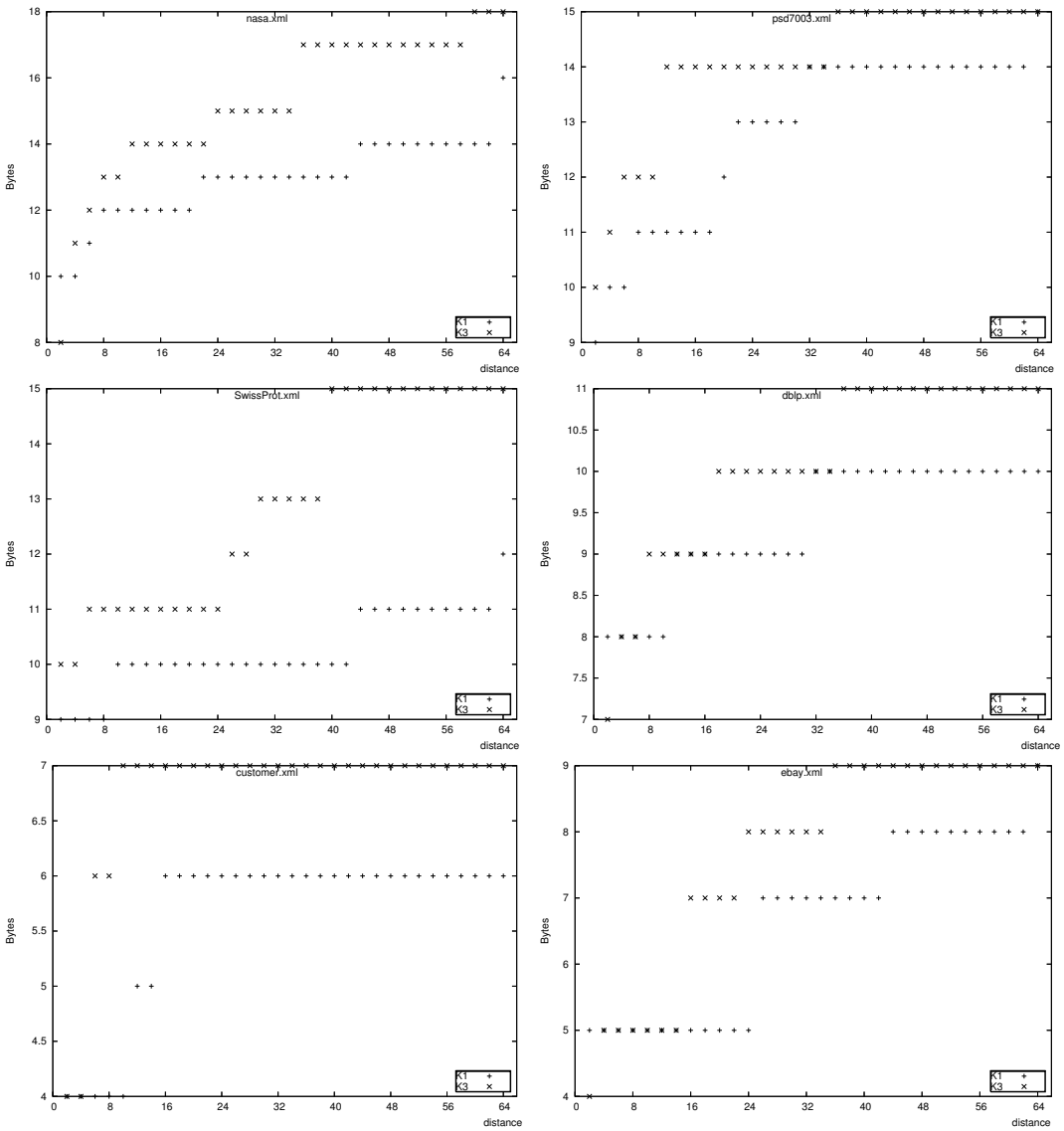
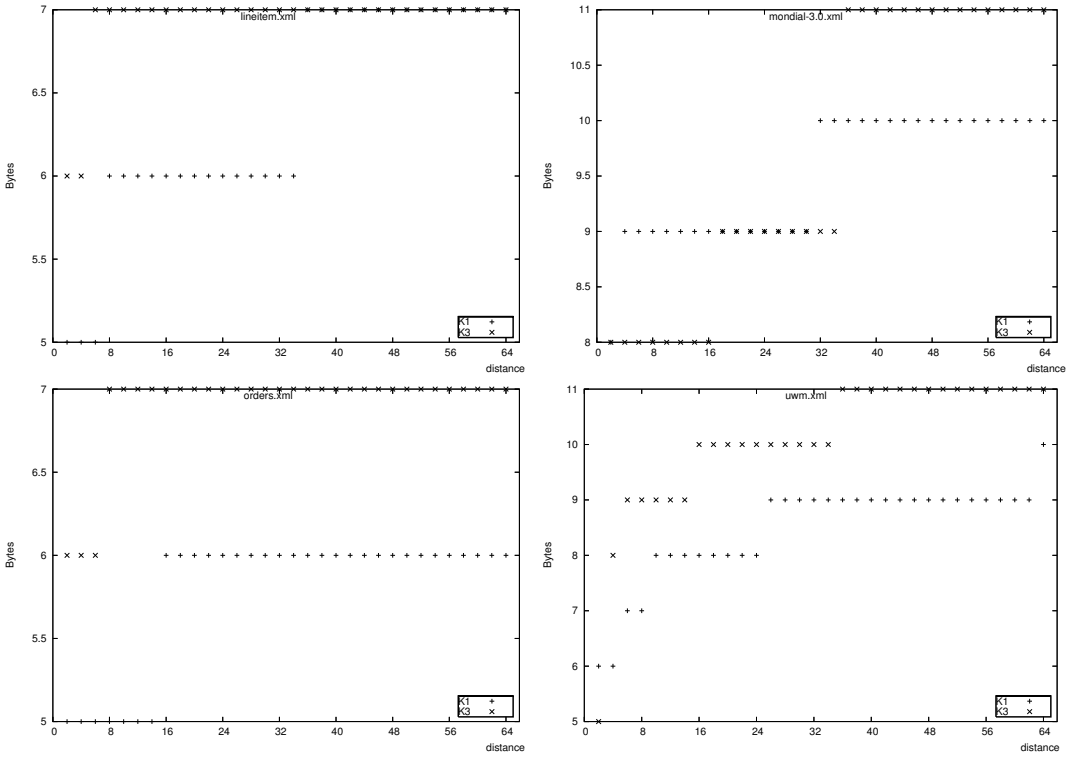


**Abbildung B.2** K1 im Vergleich zu K3: maximale DeweyID-Länge



# Literaturverzeichnis


[1] M. Haustein
*Eine XML-Programmierschnittstelle zur transaktionsgeschützten Kombination von DOM, SAX und XQuery*
11. GI-Fachtagung Datenbanksysteme in Buisness, Technologie und Web (BTW), Karlsruhe, Deutschland, S. 265-284 (März 2005)

[2] T. Härder, E. Rahm
*Datenbanksysteme - Konzepte und Techniken der Implementierung*
Springer-Verlag (2001)

[3] M. Haustein, T. Härder:
*taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API*
Proc. 7th ADBIS Conf., Dresden Germany, S. 88-102 (2003)

[4] M. Wagner
*Die Dewey-Dezimalklassifikation in XML-Datenbanken*
Projektarbeit TU Kaiserslautern (2005)

[5] D. Comer:
*The Ubiquitous B-Tree*
ACM Comput. Surv., Vol 11, No 2, S. 121-137 (Juni 1979)

[6] P. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury
*ORDPATHSs: Insert-Friendly XML Node Labels*
ACM SIGMOD Industrial Track, S. 903-908 (2004)

[7] M. Haustein, T. Härder
*Adjustable Transaction Isolation in XML Database Management Systems*
Proc. 2nd Int. XML Database Symposium, Toronto, Canada, Aug. 2004, LNCS 3186, Springer, S. 173-188 (2004)

[8] M. Haustein, T. Härder
*Optimizing Concurrent XML Processing*
submitted (2005)

[9] K. Luttenberger
*Sperrprotokoll in XML-Datenbanksystemen auf XML-Dokumenten*
Diplomarbeit TU Kaiserslautern (Juni 2005)

[10] D. Florescu, D. Kossmann
*A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database*
Rapport de Recherche, No. 3680, INRIA, Rocquencourt, France (1999)

[11] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, C. Zhang
*Storing and Querying Ordered XML Using a Relational Database System*
Proc. ACM SIGMOD, Madison, Wisconsin, S. 204-215 (Juni 2002)

[12] M. Yoshikawa, T. Amagasa
*XRel - A Path-Based Approach to Storage and Retrieval of XML Documents using Relational Databases*
ACM Transactions on Internet Technology 1(1), S. 110-141 (Aug. 2001)

[13] Gerome Miklau
XML Data Repository
University of Washington, http://www.cs.washington.edu/research/xmldatasets

[14] Webseite von DBLP.xml: http://dblp.uni-trier.de/xml

[15] J. X. Yu, D. Luo, X. Meng, H. Lu:
*Dynamically Updating XML Data: Numbering Scheme Revisited*
Word Wide Web: Internet and Web Inf. Systems, 8, S. 5-26 (2005)

[16] T. Böhme, E. Rahm:
*Supporting Efficient Streaming and Insertion of XML Data in RDBMS*
Proc. 3rd. Int. Workshop Data Integration over the Web (DIWeb), Riga, Latvia, S. 70-81 (2004)

[17] *Document Object Model (DOM) Level 2 / Level 3 Core Specification*
W3C Recommendation (Nov. 2000 / Apr. 2004)

[18] D. Brownell:
*SAX2*
O'Reilly-Verlag (2002)

[19] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, J. Siméon
*XQuery 1.0: An XML Query Language*
W3C Working Draft (2004)

[20] *Extensible Markup Language*
http://www.w3c.org/XML