Entwurf und Implementation des TextArray auf Sekundärspeicher im Rahmen des XEE Projektes

Guido Draheim
<draheim@informatik.hu-berlin.de>
- Studienarbeit -
Januar 2003

Projektziel der "XML Query Execution Engine" (XEE) ist es, ein prototypisches Anfrage- und Verwaltungssystem für XML-Dokumente zu entwerfen und zu implementieren. Den Kern des Systems bildet die eigens entworfene Datenstruktur des Access Support Tree/ TextArray (AST/TA). Im AST/TA Modell verwaltet das TextArray den Teil des Textinhalts von XML-Dokumenten. Die vorliegende Arbeit widmet sich dem Entwurf und der Implementierung des TextArrays für blockorientieren Sekundärspeicher, wobei auf die bekannte Technik der Positions-B^+-Bäume zurück gegriffen wird. Die Arbeit führt an das Modell der AST/TA-Datenstruktur heran, und betrachtet im XML Umfeld typisch zu erwartende Such- und Veränderungs-Operationen. Die Operationen werden in Hinblick auf Ansätze zur Textspeicher-Implementation betrachtet. Hierbei zeigt sich die besondere Eignung von Positions-B+-Bäumen zur Verwaltung des Textspeichers. Diese Eignung wird mit ersten Messungen der tatsächlichen Implementation gestützt. Abschliessend wird ein Ausblick gegeben, wie ein TextArray für den AST/TA Ansatz erweitert werden kann, und stützen somit die Entscheidung für eine spezialisierte Implementation des TextArrays im Rahmen des XEE Projektes.

Inhaltsverzeichnis

(1) Einleitung

Das XML Format hat sich als Standard für die Darstellung semi-strukturierter Daten etabliert. Es eignet sich für Operationen des Information Retrieval und Datenbanken und schafft eine Verbindung dieser Bereiche. Im Information Retrieval mit ihren dokument-abhängigen Strukturen werden die Auszeichnungen viel für die Sammlung von Metadaten genutzt. In Datenbanken sind die Strukturen regelmäßiger, das XML Auszeichnungssystem wird zumindest als Austauschformat verwendet. Die bisherige Integration von freien XML Texten erfolgt zumeist durch Datenbanken mit angeschlossener Suchmaschine, die speziell auf diese Speicherform angepasst wurden.

Die vorliegende Arbeit ist eingebettet in das XEE Projekt (XML Query Execution Engine) [XEE]. Der darin verfolgte AST/TA Ansatz trennt XML Dokumente in einen hierarchischen Auszeichnungsbaum (Access Support Tree, AST) und den reinen Textinhalt zwischen den XML Elementen. Beide Teile werden getrennt auf einem Sekundärspeicher gespeichert. Der reine Textinhalt ist ein logisch linearer Textspeicher und wird als TextArray (TA) bezeichnet. Auf diesen können bekannte lineare Such-Operationen für Textinhalte vereinfacht abgebildet werden, da die XML Auszeichnungen nicht beachten werden müssen.

Die Trennung ermöglicht allgemein, Such-Operationen und Veränderungs-Operationen weitgehend getrennt zu betrachten, sowohl im linearen Textinhalt als auch im hierarchischen Strukturbaum, und jeweils bekannte Methoden einzusetzen. Die Trennung von XML Textinhalt und Strukturhierarchie wurde als Modell schon betrachtet, etwa im Proximal Nodes Modell, und Vorteile bei Such-Operationen aufgezeigt [PN]. Im AST/TA Ansatz wird dies erweitert auf Veränderungs-Operationen [ASTTA]. In diesem Rahmen des XEE Projektes wird hierzu eine Speicherung des TextArray benötigt, dass auch Veränderungen im Textinhalt effizient unterstützt.

Die Implementation des TextArray soll an dieses Modell angepasst sein. Sie findet zu einer gegebenen linearen logischen Position schnell die zugehörige physikalische Stelle auf einem blockorientierten Sekundärspeicher und die Stellen nachfolgender Positionen. Veränderungs-Operationen sind effizient möglich durch nichtlineare physische Anordnung der Blöcke, und durch den Leerraum in den Blöcken zum Einfügen/ Überschreiben/ Löschen von Textbereichen bis zu einzelnen Zeichen. An diese Methodik wird in einem eigenen Abschnitt dieser Arbeit herangeführt.

Im abschließenden Teil der Arbeit wird an erweiterte Formen einer Implementation des TextArray im Rahmen des XEE Projektes herangeführt. Dabei werden auch erweiterte Operationen des AST/TA betrachtet und deren Unterstützung durch eine spezialisierte Implementation des TextArray. Dies ist jedoch weiterer Forschung vorbehalten, und wurde im Implementationsteil dieser Studienarbeit nicht angefangen.

1. AST/TA und Umfeld

Dieser Abschnitt ist stark zentriert auf die Verwendung von XML. Betrachtet werden insbesondere Methoden des Information Retrieval, und Verwendung von XML im Bereich der Datenbanken. Die resultierenden Anforderungen an Veränderungs-Operationen sind unterschiedlich, und sollen mit dem AST/TA Modell begünstigt werden. Das TextArray ist Teil des Modells zur Verwaltung der Urdaten auf dem Sekundärspeicher.

(2) XML Entwicklung und Einfluss

XML entstammt einer langen Entwicklung zur Speicherung und Durchsuche von Dokumenten, ursprünglich als SGML Dokumente [SGMLH]. Dabei wird einfacher Text, wie er dem Menschen durch Inhalt und Anordnung regelmäßig verständlich ist, um Auszeichnungen erweitert. Diese Auszeichnungen sind Metadaten, syntaktisch vom Textinhalt getrennt, die in Suchabfragen über den Text einbezogen werden können. Die Metadaten erlauben eine verfeinerte semantische Interpretation der mehrdeutigen Darstellung des reinen Textinhalts. SGML hat so eine jahrzehntelange Anwendung im Umfeld des (textuellen) Information Retrieval gefunden, ein Anwendungsfeld in dem Anfragen mit möglichst passenden Textstellen beantwortet werden.

Die zusätzliche Auszeichnung von Text mit erweiterten Attributen wurde später auch in anderen Anwendungen verwendet, bei der es keine Aufgabe war, als semantische Hilfestellung für das Information Retrieval zu dienen, sondern nur die schon entwickelten Hilfsprogramme und Funktionsbibliotheken wiederverwendet wurden. Hervorzuheben ist hier HTML, das aus der SGML Entwicklung abgeleitet wurde als Darstellungsformat für Texte, und das als wesentliches Element im WWW zum Austausch und Verbreitung von Informationen im Internet genutzt wird [WWW]. Die Auszeichnung von Textstücken dient der verfeinerten Darstellung des Textinhaltes und kann nur sehr beschränkt Interpretationshinweise zu inhaltlichen Bedeutungen liefern.

Die fehlende Erweiterbarkeit des HTML Auszeichnungssystems und die Kritik dieses Zustandes durch SGML Anwender führte zur Entwicklung von XML. Das XML Format trennt die Darstellungsform vom Auszeichnungssystem, und hat nachfolgend die Verwendung von SGML in vielen Bereichen des klassischen Information Retrieval verdrängt, da es funktionell alle Möglichkeiten von SGML in vereinfachter Form bereitstellt [W3XML]. Die XML-formatierten Dokumente können durch die vereinfachte Form leicht als Austauschformat verwendet werden, und durch ihre weitreichenden Darstellungsformate soll es auch das HTML-Format im WWW verdrängen.

Auf der Basis von XML entstehen Systeme, die spezialisierte Auszeichnungen ermöglichen, die den jeweiligen Daten angepasst sind, und die Speicherung und den Austausch zwischen Systemen unterstützen. Datenbestände wurden im WWW bis dahin mit HTML visualisiert, man konnte Daten von dieser Darstellungsform jedoch nur schwer wiedergewinnen. XML ermöglicht, die Struktur der Datenbestände direkt zu übernehmen und mit Darstellungsregeln nur zu verknüpfen [CSS2]. Dies verändert die Form des Datenaustausches, da die Serialisierung in XML sowohl zur visuellen Darstellung als auch zur reinen Datenübertragung zwischen Systemen dienen kann.

(3) Der Aufbau von XML Dokumenten

Ein XML-Dokument besteht aus drei Teilen, wobei beide Deklarationen weggelassen werden können, wenn sie der Anwendung bekannt sind:

(1)XML-Deklaration

(2)Dokument-Typ-Deklaration (mit DTD, *document type definition*)

(3)XML-Dokument-Instanz

Die hier angegebene Reihenfolge muss dabei eingehalten werden. Die XML-Deklaration kommt zuerst, da sie die Version des XML-Formats angibt und die Zeichenkodierung des Dokumentes bestimmt, und so die Wahl der Einlese-Routinen. Die Zeichendarstellung eines XML-Dokumentes (etwa als Datei in einem Sekundärspeicher) erhält so eine definierte Form, die schreibende und lesende Anwendung entkoppelt. Die Anwendung kann eine eigene interne Darstellung wählen, etwa bezüglich der Zeichendarstellung oder der Form der Darstellung des Inhaltes in Teil (3). In der Regel werden hier Hilfsbibliotheken verwendet und die Anwendung kann deren Einlese-Routine die erwartete Form der XML-Darstellung dann angeben. Die Routinen der Hilfsbibliotheken bewirken dann die transparente Umsetzung von der vorgefundenen externen XML-Darstellung.

Die Dokument-Typ-Deklaration beschreibt das Schema und die erwartete Struktur der nachfolgenden Dokument-Instanz, die den eigentlichen Inhalt des Dokumentes repräsentiert. Eine Anwendung kann selbst Kenntnis über die Struktur haben und benötigt dadurch keine explizite Angabe im Dokument. Dagegen können Hilfsbibliotheken verwendeten werden, die mit der DTD die erwartete Struktur prüfen können (*Validation*), bevor die Anwendung darauf zugreift, und können so wiederum einen Beitrag leisten, dass schreibende und lesende Anwendung entkoppelt werden [DT4DTD]. Die DTD kann auch von der Anwendung den Hilfsbibliotheken direkt übergeben werden und dadurch eine Prüfung durch deren Routinen bewirken, ohne dass DTD Angaben im XML-Dokument erscheinen.

Der eigentliche Inhalt des XML-Dokumentes folgt nach. Die Struktur dieser Daten wird dabei nicht in der Anwendung vorgegeben, sondern anhand von Markierungsdaten (sogenannten „tags“) erkannt, die in die Dokument-Darstellung eingebettet sind. Die „tags“ erlauben mittels einer Anfangs-Markierung und einer Ende-Markierung die Abgrenzung eines Datenbereiches von anderen Bestandteilen des Dokuments. Die Markierung erfolgt mittels eines Namens in spitzen Klammern, und über eine DTD können Datenbereichen mit diesem Namen genauere Bedeutungen zugeordnet werden. Der Gesamtinhalt wird ebenfalls durch eine Markierung abgegrenzt, der sogenannten ROOT-Markierung, und trennt es von anderen Bestandteilen des XML-Dokumentes wie den vorstehenden Deklarationen.

```
<?xml version=“1.0“ encoding=“UTF-8“ ?>
<!DOCTYPE book SYSTEM “book.dtd“>
<book>
  <author>Wilhelm Busch</author>
  <title>Max und Moritz</title>
</book>
```

Die DTD kann wie hier als externe Datei „book.dtd“ angegeben werden, oder deren Daten direkt in das XML-Dokument eingeschrieben werden (oder deren Daten aus der Applikation selbst kommen). Es gibt eine Reihe von Standard-DTDs, die den Datenaustausch stark vereinfachen, da deren Namensangabe genügt, um den XML-Leser

darauf zugreifen zu lassen, ohne deren Daten zu duplizieren, etwa als dem XML-Dokument beigefügte DTD-Datei oder als Zeilen im XML-Dokument. Hierbei wird statt SYSTEM wie im Beispiel die Angaben „PUBLIC *name*“ verwendet.

Ein XML-Dokument darf noch eine Reihe anderer Teile enthalten, die nicht dem Textinhalt zugeordnet werden, etwa Kommentare und „Processing Instructions“, auf die hier nicht weiter eingegangen wird. Bedeutend für das Finden und Bewerten von Textstellen (Information Retrieval, siehe nachfolgenden Abschnitt) sind die Attribut-Angaben. Diese erscheinen innerhalb der Anfangs-Markierung nach dem eröffnenden Namen in einem speziellen Format, und sind logisch eine Abbildung von Attribut-Namen auf Werte.

Jedes dieser Attribute (unter einem Namen) darf höchstens einmal auftreten. Eine DTD kann für verschiedene Markierungen auch Attribute vorgeben – beispielsweise können sie als „#required“ als erforderlich bezeichnet werden oder mittels „#implied“ unter einem bestimmten Wert angenommen werden, auch wenn sie nicht explizit in der XML-Darstellung auftauchen. Durch Attribute kann die gleiche Strukturierung des Textinhaltes mit verschiedenen erweiterten Bedeutungen und Hinweisen versehen werden, die von der Anwendung abgefragt werden können.

<author>Wilhelm Busch</author> schrieb <title>Max und Moritz</title>

Und mit zusätzlichen Informationen in den Auszeichnungen:

<author from=“1832“ to“1908“>Wilhelm Busch</author> **...**

... schrieb <title published=“1895“>Max und Moritz</title>

(4) Zuordnung von Metadaten zu Textstellen in XML

In der XML Struktur hat jede Anfangs-Markierung genau eine Ende-Markierung, die einen Textinhalt auszeichnen, der zwischen beiden Markierungen liegt. Der begrenzte Bereich kann selbst wieder XML Auszeichnungen enthalten, und der ausgezeichnete Bereich kann auch von anderen XML Auszeichnungen eingeschlossen werden. Eine Textstelle kann daher im abgegrenzten Bereich mehrerer XML Auszeichnungen liegen. Es lässt sich eine Operation definieren, die genau diese Menge von Auszeichnungen zu jeder Textstelle des XML Dokumentes findet.

Der Umfang der gefundenen Auszeichnungen je Textstelle ist dabei nicht beschränkt, zumindest ist jede Stelle des Textinhalts von der einen Auszeichnung umschlossen, die nach der Struktur von XML Dokumenten erforderlich ist, um den Textinhalt von vorstehenden Deklarationen zu trennen. Neben der Einordnung in Abgrenzungen finden sich weitere Werte in den Attributen der konkreten (d.h. aktuell im Dokument physisch vorliegenden) Markierungen, die als konkrete Metadaten einer Textstelle zugeordnet werden. Über Verknüpfungs-Operationen können diesen konkret lesbaren Metadaten einer XML Datei weitere Informationen zugeordnet werden, etwa die in einer DTD angegebenen "#implied" Werte.

Die Struktur der Zuordnung von Metadaten ergibt sich aus dem Dokument selbst und ist nicht für bestimmte Positionen im Textinhalt im Vorhinein vorgegeben. Die Kategorisierung von Textstellen durch die Metadaten ergibt sich immer aus den konkreten Auszeichnungen und deren Attributen. Algorithmen für Suchanfragen können auf diese konkreten Strukturen und Metadaten bezug nehmen für Einschränkungen und logische Verknüpfungen.

(5) XML im Data Retrieval und Information Retrieval

Zur Beschreibung der Aufbereitungsprozesse im Information Retrieval im nächsten Abschnitt gehen wir hier kurz auf die Suchprozesse ein. Unter Information Retrieval (IR) versteht man jede Art des Wiederauffindens von Informationen (in maschinell gespeicherten Daten) zu thematisch-inhaltlichen Fragen. In seiner allgemeinen Auffassung benötigt die IR keine vorgegebenen Strukturen der Daten, auch wenn die weitverbreitetste Form von IR die „Inhaltliche Suche in Texten“ ist. Im IR wird sorgfältig zwischen Informationen, Daten und Wissen sowie verschiedenen Sichtweisen auf Daten (logische, semantische, anordnungs) unterschieden [Fuhr'IR].

Im Information Retrieval werden Informationssysteme in Bezug auf ihre Rolle im Prozess des Wissenstransfers vom (menschlichen) Wissensproduzenten zum Informations-Nachfragenden betrachtet. Dabei können Anfragen vage formuliert sein und die verfügbaren Daten nur ungefähr passen. Der Zusammenhang der (menschlichen) Anfrage zu den existierenden (maschinenlesbaren) Daten wird über eine Bewertung betrachtet.

Zur Erleichterung des Auffindens und Bewertens von Daten können Strukturen und Prozeduren eingesetzt werden, die heute unter dem Begriff des Data Retrieval (d.h. Fakten Auffinden) subsumiert werden. Diese suchen nach konkret vorhanden Merkmalen in Datenmengen. Bezüglich des XML Formats ist es möglich, in die Suchanfragen die Metadaten der XML Auszeichnungen einzubeziehen [KL'XQ]. Diese Suchanfragen gehen über reines Text Retrieval hinaus, dass sich auschliesslich auf den Textinhalt bezieht. Zur Abgrenzung von Data Retrieval und Information Retrieval diene folgende Tabelle nach van Rijsbergen [VR'IR'79].

	Data Retrieval	***Information Retrieval***
Matching	Exact Match	Partial Match, Best Match
Inference	Deduction	Induction
Model	Deterministic	Probalistic
Classification	Monothetic	Polythetic
Query Language	Articificial	Natural
Query Specification	Complete	Incomplete
Items wanted	Matching	Relevant
Error Response	Sensitive	Insensitive

Die Suchprozesse des Data Retrieval ergeben dabei eine Menge von Kandidaten, die für das Ergebnis des maschinellen Information Retrieval Prozesses bewertet werden. Üblich ist dabei ein eindimensionales Maß als Ergebnis, dass eine Rangfolge der gefunden Kandidaten angibt. Diese Rangbildung kann Werte aus den konkreten Fundstellen einbeziehen, dazu gehören etwa Abstände und Längen von Textbereichen. Es werden außerdem verschiedenen Kategorisierungen aus den Metadaten verschiedene Maßwerte

zugeordnet werden, und Attribute in den zugeordneten Auszeichnungen können selbst Wertangaben enthalten, die genutzt werden können. Diese diskreten Werte werden in Berechnungen verknüpft zum Ergebnisrang der Kandidaten, die damit sortiert werden.

Das XML Format ermöglicht, dass die gefundenen Werte über einem Text im XML Dokument selbst abgespeichert werden können, sodass die Bewertung wieder für nachfolgende Prozesse verwendet werden kann. Ein folgende Suchanfrage kann dann einschränken nur jene Stellen zu finden, die einen bestimmten Bereich des Wertmaßes in den Attributen abgespeichert haben. Dies ist dann ein deterministischer Prozess des Data Retrieval selbst, der die zu verarbeitenden Menge von Fundstellen stark eingeschränkt.

(6) Erkennung und Speicherung von Mustern in XML

Wir betrachten hier die Suche und Bewertung von Informationen in Textdokumenten im Rahmen des Information Retrieval, und betrachten die Aufbereitungsprozesse für Texte. Der ursprüngliche Text kann leicht in ein XML Dokument gewandelt werden, indem Anfang und Ende mittels XML-Markierung ausgezeichnet wird, und so die ROOT-Markierung bildet wie vor beschrieben im Abschnitt zum Aufbau von XML Dokumenten. Textdokumente bestehen typisch aus Worten und trennenden Leerzeichen sowie anderen ebenfalls trennenden Zeichen (oft als Trennzeichen bezeichnet).

```
<book>
  Wilhelm Busch schrieb Max und Moritz im Jahre 1895.
  Frank Herbert's erstes Buch war Der Wüstenplanet.
</book>
```

Eine typische Anfrage in einem Information Retrieval Prozess kann eine Data Retrieval Suche nach bestimmten Worten im Text auslösen, deren Fundstellen dann bewertet werden. Dies kann nicht nur einfache Worte verwenden, sondern auch nach Klassen einordnen – so deutet in obigem Text die Großschreibung auf Eigennamen in. Die mögliche Bewertungsformel kann einschließen, ob die gefundenen Teile im gleichen Abschnitt, gleichen Satz auftreten oder durch „und“ verbunden sind. Zur Unterstützung dieser Formulierung kann eine Mustererkennung entsprechende Strukturen erkannt haben und im XML Dokument als Markierung abgespeichert haben.

```
<book>
  <phrase>Wilhelm Busch schrieb Max und Moritz im Jahre 1895.</phrase>
  <phrase>Frank Herbert's erstes Buch war Der Wüstenplanet.</phrase>
</book>
```

Die Mustererkennung kann etwa in Betracht ziehen, ob Satztrennzeichen existieren, ob Zeilenumbrüche vorliegen oder Leerzeilen eingefügt sind. Durch explizite Hinzunahme von XML-Markierungen kann von bestimmten Textinhalten abstrahiert werden, etwa ob Leerraum durch Leerzeichen oder Zeilenumbruch entsteht. Nach der <phrase> Erkennung etwa kann angenommen werden, dass sich der Textinhalt des Dokumentes semantisch nicht unterscheidet, egal welche Form von Leerraum zwischen den <phrase>n liegen, und ob Zeilenumbrüche enthalten sind, die die <phrase> Erkennung nicht beeinflusst hätten (Äquivalenzklasse).

Eine weitere Fragestellung kann etwa beinhalten, ob Publikationsinformationen in einem Text auftreten. Durch entsprechende Hilfsmittel des Information Retrieval kann etwa bewertet werden, dass die Muster [name]-schrieb-[name] und [name]'s-Buch-[sei]-[name] jeweils auf Publikationen hindeuten. Zur späteren Erleichterung des Auffindens von Publikationsinformationen kann dieses Resultat durch Markierungen im XML-Dokument vermerkt werden. Dieser Prozess kann fortgesetzt werden und so Teile des Textes mit Markierungen wie etwa <author> und <title> versehen.

```
<book>
  <phrase><author>Wilhelm Busch</author> schrieb
  <title>Max und Moritz</title> im Jahre <year>1895</year>.</phrase>
  <phrase><author>Frank Herbert</author>'s erstes Buch war
  <title>Der Wüstenplanet</title>.</phrase>
</book>
```

Neben der Erkennung von Mustern im Textinhalt und der Markierung der Muster kann ein solcher Bewertungsprozess auch die Fundstellen verknüpfen und die gewonnen Werte wiederum speichern. Im Beispiel kann die Erkennung der Jahreszahl etwa an den Buchtitel geknüpft werden und im XML-Dokument als Attribut des <title> eingefügt werden. Verknüpfungen mit Daten müssen dabei nicht auf das Dokument und dessen Inhalt beschränkt bleiben. Das Erscheinungsdatum des zweiten Eintrags kann auch aus einer Datenbank kommen und zur beschleunigten Bewertung von Fundstellen hier eingefügt werden.

```
<book>
  <phrase><author>Wilhelm Busch</author> schrieb
  <title published="1895">Max und Moritz</title>
     im Jahre <year>1895</year>.</phrase>
  <phrase><author>Frank Herbert</author>'s erstes Buch war
   <title published="1965">Der Wüstenplanet</title>.</phrase>
</book>
```

Dabei sind die Attribute der XML-Markierungen nicht Teil des eigentlichen Textinhaltes, sondern dienen als zusätzliche Daten der Beschleunigung von Prozessen der Suchanfragen und Bewertung der Fundstellen. So kann für eine Fundstelle wie „Max“ danach gefragt werden, ob es sich um eine Zeitschrift oder ein Buch handelt, und ob es in den Zeitkontext einer möglichen IR Anfrage passt. Das XML Format stellt dazu Möglichkeiten für das Data Retrieval bereit, die im Umfeld von SGML und der Forschung zu IR Modellen und unterstützenden Implementationen entstanden sind. Der ursprüngliche Textinhalt bleibt dabei unangetastet. Er kann vollständig durch Entfernung der XML-Markierung wiederhergestellt werden.

(7) XML Serialisierung von Datenbanken

Die XML Auszeichnung kann auch zur serialisierten Darstellung der Struktur und des Inhalts von Datenbanken eingesetzt werden, wobei die Auszeichnungen die Begrenzung der Felder und Datensätze in der Datenbank angeben. Der Einsatz der XML Serialisierung als Transfer-Format ist verbreitet, auch wenn die Datenbank selbst intern eine Speicherung nutzt, die nicht auf XML basiert. Die Strukturierung der Daten in Datenbanksystemen erfolgt in der Regel außerhalb des Datenspeichers, die Felder benötigen hierbei keine Begrenzer um die Feldinhalte im Speicherformat. Die Begrenzer werden bei der Serialisierung hinzugefügt.

Bei der Generierung eines XML Transferdokumentes aus einer relationalen Datenbank bietet es sich an, die Feldnamen als Namen für die innerste Auszeichnung zu verwenden und die Datensätze selbst auszuzeichnen zur Gruppierung der Felder [XML'RDB]. Die Zugriffsstruktur der Datenbank wird so in XML Auszeichnungen umgesetzt und der Inhalt der Datenbank erscheint als Textinhalt des XML Transferdokumentes. Es handelt sich um eine Serialisierung der Datenbank, die für alle Datenbanken möglich ist, einschließlich der hierarchischen Datenbanken, die sich ebenso einfach in die hierarchische Grundstruktur von XML Dokument abbilden lassen.

Datenbank (mit 3 Datensätzen)

Wilhelm Busch	Max und Moritz	1895
Frank Herbert	Der Wüstenplanet	1965
Stephen Hawking	Raum und Zeit	1996
<author/...	<title/...	<year/....

Serialisiert mit Leerraum um die Auszeichungen:

```
<record><author>Wilhelm Busch</author>
  <title>Max und Moritz</title> <year>1895</year>
</record>
<record><author>Frank Herbert</author> .....
```

Die serialisierte Darstellung von Datenbanken als XML Dokumente weist eine sehr regelmäßige Form auf. Abfragen über der Struktur von Datenbanken können direkt umgesetzt werden in Strukturbedingungen bezüglich der XML Auszeichnungen[XPath]. Ebenso können Anfragen an den Feldinhalt von Datenbanken auch auf das XML Format übertragen werden, und beziehen sich hier auf Textbereiche in einem ausgezeichneten Abschnitt des Dokumentes. Die Auffindung der Daten im serialisierten Format des XML ist dabei jedoch langsamer, wenn direkt im linearen XML Dokument gesucht wird.

(8) XML Einbettung in Datenbanken

Neben der daten-zentrischen Verwendung von XML steht die dokumenten-zentrische Anwendung, bei der dem XML Dokument kein Datenbankschema bei der Generierung zugrunde lag. Die Struktur von XML Dokumenten kann dabei unregelmäßig und tief geschachtelt sein, sodass sich die hierarchische Form der XML Auszeichnungsstruktur nicht direkt in ein Datenbankschema abbilden lässt. Eine Verwendung einer Datenbank bietet jedoch optimierte Zugriffsmechanismen auf die Struktur der Dokumente. Als verbreitete Lösung verwendet man die Erweiterung klassischer Datenbanken, die die Struktur von XML teilweise in Datenbankstrukturen umsetzt, und kleinere Teile in Feldinhalten speichert. Das Abfragesystem der Datenbank wird dabei erweitert und kann dann auch auf die XML Strukturen in den Feldinhalten bezug nehmen [DB2XML]/ [ORA9XML]. So können auch unregelmäßige XML Dokumente in üblichen Datenbank-systemen gespeichert werden.

Ein Abfragesystem für XML Dokumente verwendet vielfach XPath Ausdrücke, und die XML Erweiterung der Datenbanken erlaubt hier, Teile der Strukturanfrage auf das Datenbankschema zu übertragen. Man geht dann über die Auswahl an Feldgrenzen der Datenbank hinaus, und übernimmt nur jene Teile in das Ergebnis, die einem mit XML tags ausgezeichneten Abschnitt in den Feldinhalten entnommen sind. Dieses erfordert jedoch,

dass nach der anfänglichen Auswahl von Felder der Datenbank auch der Feldinhalt geprüft werden muss, ob Teile den XPath Ausdruck erfüllen [XPath]. Dies erfordert das Lesen der Daten in den Feldern selbst, und geht über den optimierten Zugriff auf die Verwaltungsstruktur der Datenbank hinaus.

Zur Umsetzung von dokumenten-zentrischen XML Daten in Datenbankschema haben sich verschiedene Methodiken herausgebildet, auf die hier nicht eingegangen werden soll. Zur effizienten Nutzung braucht es XML Texte, die Bereiche mit regelmäßigen Aufbau enthalten, die dann in traditionelle Zugriffsstrukturen von Datenbanken umgesetzt werden, und darin enthaltenen unregelmäßigen Anteilen mit XML Auszeichnungen, die als Feldinhalte gespeichert werden. Das Beispiel unterteilt nach Satzaussagen und nimmt den Autor als Schlüssel mit weitergehenden Hinweisen als XML Text.

Datenbank “publishdata“

Wilhelm Busch | schrieb <title year=1895>Max und Moritz</title>
Frank Herbert | erstes Buch <title year=1965>Der Wüstenplanet</title>
Stephen Hawkings | weiter in <title year=1996>Raum und Zeit</title>

<author/.. <bookinfo/...

Dieses Beispiel zeigt, dass die eindeutige Rekonstruktion eines Textdokuments aus solchen Datenbanken schwer ist, etwa wenn die Autor-Information im ursprünglichen Text am Ende des Satzes stand, was durch die Grammatiken vieler natürlicher Sprachen ermöglicht wird. Man kann sich zusätzliche Informationen zur Rekonstruktion abspeichern, etwa in einer eigenen Spalte der Datenbank, womit aber wieder Strukturdaten in die Datenbank eingebettet werden, die zur Suche und Veränderung im Textinhalt selbst nicht benötigt werden [DB2XML-]/[DB2XML+].

(9) Veränderungs-Operationen auf XML-Dokumenten

Bei einer Veränderungs-Operation in XML-Daten wird ein Bereich anhand der XML Auszeichnung und Attribute selektiert. Die Daten des selektierten Bereiches können dann verändert werden. Es gibt mehrere verschiedene Veränderungs-Operationen, die einfachste ist der Austausch des Textinhaltes in einem Endknoten der hierarchischen XML Struktur (text update). Handelt es sich um eine daten-zentrische XML Speicherung, so entspricht dies der Veränderung des zugeordneten Feldes einer Datenbank, die über das zugehörige Datenbankschema zur XML Struktur selektiert werden kann.

XML Dokumente können auch verändert werden, indem Auszeichnungen und Attribute hinzugefügt und gelöscht werden, etwa im Rahmen der Aufbereitung von Dokumenten für das Data Retrieval, wie im Abschnitt (6) zur Speicherung von Mustern in XML (markup update). Bei der Einbindung von XML in Datenbanken muss die Strukturänderung des Dokumentes im System nachvollzogen werden. Diese Anpassungen werden in dieser Arbeit nicht betrachtet, das veränderte Dokument muss jedoch gültig bleiben zur Übertragung in das Datenbanksystem und dessen Zugriffsschema. In der XML Darstellung

der Datenbank sind die veränderten Teile je Bereiche eines größeren Dokuments, die hierarchisch weitere Datenfelder enthalten können [XMLnDB].

Nimmt man die XML Dokumente in ihrer Darstellung als Textdatei, so sind alle Änderungen bezogen auf einen Textabschnitt. Verändert man die Textdatei direkt, wird jeweils der zugehörige Textabschnitt verändert, einschließlich Textinhalt und Markierungen (tags). Auch die "markup updates" fügen hierbei jeweils "tags" ein in das XML Format, sodass keine Rücksicht auf das Zugriffsschema einer Datenbank genommen werden muss. Der Zugriff selbst ist jedoch langsamer, und wird die Auswahl von Teilen durch XPath Ausdrücke ermöglicht, so muss die Zugriffsstruktur durch Lesen des Dokumentes und der enthaltenen "tags" erkannt werden (durch serielles Parsen etwa mit [SAX]).

Zum schnelleren Auffinden von Positionen bieten sich Indexverfahren an. Ein "text update" erfordert hier ein Nachführen der Positionen der "tags", wenn sich durch den "text update" die Positionen verschieben. Für große XML Dokumente braucht es eine Implementation, die Datenbank-Operationen effizient unterstützt. Neben den XML-erweiterten Datenbanken werden dazu weitere Ansätze erforscht [XMLDBs], insbesondere "native XML Datenbanken" bei denen die hierarchische XML Auszeichnung und das Datenbankschema zusammengeführt sind, und kein Nachführen erforderlich ist.

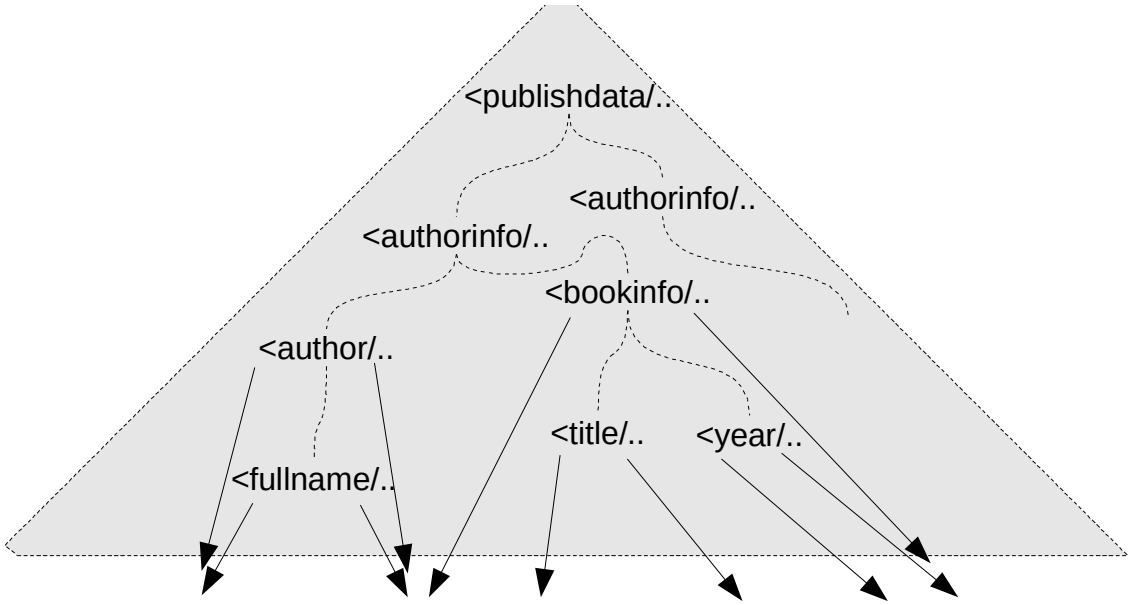
In nativen XML Datenbanken wird erreicht, dass beliebige Strukturen einer XML Datei abgebildet werden können, und sowohl die zugreifbaren Bereich schnell in der Speicherung aufgefunden werden können, als auch Änderungen unterstützt werden. Werden zur Auswahl XPath Ausdrücke verwendet, müssen dessen Angaben sich schnell abgleichen lassen zum Auffinden von Bereichen. Die Änderungen der selektierten Bereiche muss sowohl "text updates" wie "markup updates" unterstützen, neben den Änderungen des Textinhaltes also auch die Auszeichnung des Dokumentes.

(10) AST/TA Ansatz

Der AST/TA Ansatz trennt den Textinhalt eines XML Dokumentes von den Auszeichnungen der Bereiche und speichert beide getrennt [ASTTA]. Die reine Textinformation bezeichnen wir hier als TextArray (abgekürzt TA), und ist ein streng linearer Datenbereich. Eine Unterscheidung der XML Daten in regelmäßige Strukturen ist hier nicht notwendig, da die Struktur des Dokumentes in einen hierarchischen Zugriffsbaum abgebildet wird (AST, Access Support Tree). Über den AST kann man jederzeit die Position des zugehörigen Inhalts eines Feldes erhalten, als Position und Länge in einem linearen eindimensionalen Textspeicher. Der AST selbst enthält die Hierarchie und ist auch in der Lage, XML Dokumente zu verwalten, zu denen keine DTD existiert. Die Struktur muss auch nicht in eine andere Form klassischer Datenbank-managementsysteme (DBMS) abgebildet werden.

Die getrennte Speicherung ermöglicht eine Reihe zusätzlicher Möglichkeiten der Nutzung. So kann man zum Beispiel mehrere Auszeichnungen für den gleichen Text aufbauen und so die Nutzung für Information-Retrieval vereinfachen, da die Veränderungen nur im AST erfolgen. Andererseits verändern Veränderungs-Operationen auf den Feldinhalt allein nicht die Baumstruktur des AST und dadurch gegebene Anordnung der Felder im TextArray, womit datenorientierte Veränderungs-Operationen effizient unterstützt werden können, die sich auf eine effiziente TextArray Implementation stützt.

Im Rahmen des XEE (XML Query Execution Engine) Projektes wird die Verwendung des AST/TA Ansatzes untersucht [XEE] und eine Implementation des TextArray dazu benötigt. Die Implementation arbeitet dabei auf Sekundärspeicher und soll auch die datenorientierte Veränderungs-Operationen effizient unterstüzten. Eine Reihe von Fragen



zur Feldbestimmung und Auszeichungserweiterung des oberhalb des TA liegenden AST sind dabei nicht Teil der vorliegenden Arbeit – zur Adressierung von Markierungsknoten im AST Baum kann der existierende XPath Standard eingesetzt werden. Zur Angabe der Veränderungen an der Stelle braucht es weiterer Angaben, ein existierender Vorschlag ist etwa der xupdate Standardisierungsvorschlag [XUPDATE].

Im weiteren betrachtet diese Arbeit eine effiziente Implementation für das TextArray des AST/TA Ansatzes im Rahmen des XEE Projektes. Veränderungs-Operationen am AST werden durch andere Arbeiten beschrieben, die ebenfalls im Rahmen des XEE Projektes laufen. Durch die Trennung in Textspeicher und Strukturbaum bilden die Operationen des Suchens und Veränderns in den meisten Fällen nur auf einen der Teile ab: den AST mit der Strukturinhalten oder dem TA mit den Textinhalten.

2. TextArray auf Sekundärspeicher

Begriffe

Im ersten Teil führen wir an die Begriffe heran, die bei der Besprechung der Funktionalität und Implementation eines TextArray benötigt werden, dem linearen Textspeicher im Rahmen des AST/TA Modells.

(11) TextArray

Das TextArray bezeichnet die logische Ebene eines Textspeichers, eine lineare lückenlos aufsteigende Anordnung von gleichgroßen Textelementen (Zeichen). Angaben einzelner Textstellen erfolgen als ganzzahliger positiver Wert, dem Abstand zum Anfang des Textspeichers.

Diese logische Sicht des Textblocks ist besonders einfach und daher in der Computertechnik weit verbreitet, bei der Textelemente in Bytes (8bit Zeichen) gerechnet werden. Ebenso einfach sind Textelemente in Unicode (16bit Zeichen). In UTF-8 jedoch können Textelemente unterschiedliche Länge haben, 1 bis 4 Byte, sodass ein Abstand in Textelementen nicht mit einem Abstand in Bytes übereinstimmt. Ebenso werden im XEE Projekt Separatoren in die physische Speicherung eingefügt, die bei einer logischen Sicht in Textelementen nicht einbezogen werden [ASTTA].

Daher stützt sich die Implementation des TextArray ausdrücklich auf Byte Abstände, die gespeichert werden, und es wird durch die weiteren XEE Teilprojekte sichergestellt, dass diese Speicherpositionen jeweils auf den Anfang eines Textelementes weisen. Dies ermöglicht weiterhin die Behandlung auch verschiedener Kodierung von Textelementen mit gleicher unterliegender Speicherform.

(12) Speicherposition

Das TextArray hat für jedes enthalte Zeichenelement eine logische Textposition gegeben als dem Abstand in Bytes zum Anfang des linearen Textspeichers. Dieser ganzzahlige positive Wert muss in den physische Speicherort umgerechnet werden, äquivalent genannt Speicherposition oder physische Position.

Im Hauptspeicher handelt es sich bei einem Speicherort um eine Adresse, in den meisten Systemen ebenfalls ein ganzzahliger Wert, der in den linearen Hauptspeicherblock weist. Wenige Systeme nutzen segmentierte Adressen, bei denen eine Adresse ein Tupel aus ganzzahliger Segmentnummer und Abstand innerhalb des Segments ist. Wenn die Segmentnummern lückenlos aufsteigend liegen, stellt dieses eine lineare Ordnung dar, die es ermöglicht, zum Auffinden physischer Positionen ähnliche und einfache Prozeduren zu verwenden, wie sie für ganzzahlige lineare Adressierung genutzt werden.

Bezogen auf heutige Systeme, ist Segmentierung weiterhin in 16bit Systemen üblich, bei denen eine Adresse aus einer 16bit Segmentnummer und einem 16bit Index darin gebildet wird. Damit wird die Grenze der ursprünglichen maximale Speichergrösse überwunden, mit einem 16bit Index können sonst nur 2^{16}=64k Bytes angesprochen werden. Durch technische Fortentwicklung sind heute auch Hauptspeichergrößen möglich, die durch einen 32bit linearen Index allein nicht angesprochen werden können, also maximal

2^{32}=4G Bytes. Seit Pentium-Pro nutzen Intel-Prozessoren die Segmentierung aus 16bit Tagen um größeren Speicher ansteuern zu können, genannt PAE = “Page Adress Extension“, und mit wenigen zusätzlichen Bits erreichen sie so 2^{36} Bytes[Intel'P3].

Während man segmentierte Adressen und lineare Positionen oft einfach umrechnen kann, gilt es eine Unterscheidung zu treffen, insoweit Adressen in den Segmente wahlfrei angesprochen werden können oder nicht. Der oben genannte IntelP3 hat einen 64bit Datenbus und lädt/speichert Daten in diesen Blockgrößen. Dagegen stellt dort 2^{32} keine Beschränkung des wahlfreien Zugriffs dar, da der Prozessor 35+3 Pins für den Adressbus bereitstellt, die durch den Chipsatz des Rechners umgesetzt werden auf die tatsächlich verfügbaren Speicherbausteine [Intel'P4].

(13) Sekundärspeicher / -blöcke

Bei Sekundärspeicher ist die Unterteilung in Sektoren üblich, das sind gleichgroße Teile des physischen Speichers, die nur als als ganzes gelesen/geschrieben werden, kleinere Teile sind nicht wahlfrei anzusprechen. Die verbreitetste Sektorgröße ist derzeit 512 Bytes. Da Sekundärspeicher wesentlich langsamer ist als Hauptspeicher, werden Algorithmen mit Zielrichtung Sekundärspeicher darauf optimiert und verwenden zur Angabe von physischen Positionen eben direkt Tupel aus Sektoradresse und Abstand innerhalb des Sektors.

Je nach Anwendung gruppiert man die Sektoren weiter, damit nicht auf Sektoren logisch nachfolgender Daten gewartet werden muss. Die Anwendung optimiert seine Algorithmen hier auf Blockgrößen als Serie von Sektoren und spricht den Sekundärspeicher in Blockgrößen an, die nur als ganzes gelesen/geschrieben werden. Verschiedene Teile der Anwendung können dabei unterschiedliche Blockgrößen verwenden - die Verwaltung vereinfacht sich jedoch, wenn alle Teile die gleichen Blockgrößen verwenden. Im weiteren Text wird angenommen, dass nur eine Blockgröße verwendet wird zur Ansprache des Sekundärspeichers. Die Anwendung verarbeitet dann intern Tupel aus Blockindex und Abstand innerhalb dieses Blocks, einer Gruppen von Sektoren.

Anforderungen

Im diesem Teil führen wir an die Operationen heran, die für eine TextArray Implementation notwendig sind.

(14) Auffinden / Bereitstellung

Beim Auffinden muss zu einer logischen Textposition die physische Speicherposition gefunden werden. Das Bereitstellen beinhaltet bei Sekundärspeicher die Verfügbarmachung der Textstelle für die Applikation im Hauptspeicher (die "Holen"-Operation). Bei sehr großem TextArray kann dies lange dauern. Schon die algorithmische Umsetzung von der logischen Position in die Speicherposition kann unterschiedlich lang dauern, da diese Operation Werte auf dem Sekundärspeicher in die Berechnungen zum Auffinden einbezieht (meist dort Verzeichnisse genannt). Nach mehreren Zugriffen stellt findeet dann die letzte Anforderung an den Sekundärspeicher die eigentliche physische Speicherposition der Textstelle auf dem Sekundärspeicher, und kann sie im Hauptspeicher bereitstellen.

Auch wenn angenommen wird, dass der Sekundärspeicher wahlfreien Zugriff auf Blockgröße zulässt, so ist die Wahlfreiheit bei vielen technischen Verwirklichungen nicht ideal, also eine gleich lange Zeit für eine beliebige Anfrage einer Speicherposition bis zur Verfügbarkeit des Blocks im Hauptspeicher gegeben. Ohne ins Detail zu gehen, manche Bandlaufwerke in früheren Jahrzehnten waren über Markierungen in der Lage, jede Position direkt anzufahren, mussten aber dorthin spulen, was dauerte, auch wenn man dazu in einen speziellen Hochgeschwindigkeitsmodus geschaltet hat. Im letzten Jahrzehnt findet sich beispielgebend die CD-ROM, bei der jede Positionsänderung eine Anpassung der Drehgeschwindigkeit erfordert. Und auch die typischen Festplatten brauchen für den Wechsel zwischen Spuren länger als für den Wechsel zwischen Sektoren in der gleichen Spur.

Das Auffinden der Speicherstelle und deren Bereitstellung werden weitgehend synonym benutzt, in manchen Textpassagen unterscheidet sich erstere darin, dass sie die physische Textposition nur als Wert zurück gibt, die zweite auch diese Textposition auch tatsächlich verfügbar macht. In den meisten Fällen folgt dem Auffinden auch die Bereitstellung, das Auffinden wird in wenigen Fällen jedoch zeitlich entkoppelt oder die Bereitstellung wird auf Grund zwischenliegender veränderter Bedingungen abgebrochen.

(15) Suchen / Weitersetzen

Das Auffinden einer Speicherposition bezieht sich nur auf ein Textelement, zur Arbeit im TextArray greift man jedoch fast immer auf eine Anzahl angrenzender Textelemente zu. Hier ergibt sich die Anforderung, zu einer gefundenen Textstelle möglichst schnell auf die Speicherposition angrenzender Textelemente zu kommen und diese im Hauptspeicher bereitzustellen. Dabei soll dieses schneller sein als das Auffinden der ersten Speicherstelle.

Da der Sekundärspeicher in Blöcken angesprochen wird, stehen in vielen Fällen die angrenzenden Speicherstellen schon im Hauptspeicher bereit. Es ist leicht zu sehen, dass das Übergehen in den nächsten Block von der Blockgröße abhängt (bei vollständig genutzten 512 Byte Blöcken kann man durchschnittlich 256 Byte weitergehen, ehe ein neuer Block vom Sekundärspeicher angefordert werden muss). Überschreitet man den aktuellen Block, muss man die physische Position der nächsten logischen Textposition finden, die nicht physisch angrenzend sein muss, wie in der späteren Beschreibung der Algorithmen zu sehen sein wird.

(16) Einfügen / Löschen

Das Einfügen und Löschen beeinflusst die logische Position aller nachfolgenden Textelemente im TextArray. Wie in den Algorithmen zu sehen ist, muss dabei die physische Speicherstelle der nachfolgenden Zeichen nicht immer geändert werden, sondern man verändert die Zugriffsinformationen, die die Zuordnung von logischen zu physischen Positionen bestimmen.

In der Implementation erfordert das Einfügen von Textelementen zumindest eine Schreib-Operation von Textblöcken auf den Sekundärspeicher. Das Löschen von Textelementen kann auch dadurch erfolgen, dass man den Bereich der Löschung vermerkt und beim nächsten Zugriff (Auffinden oder Weitersetzen) überspringt. Physisch vorhandene aber logisch nicht zählende Stellen nennt man Lücke.

(17) Nutzungsgrad

Durch die Lücken wird mehr physischer Speicher benötigt als logisch Text darin gespeichert ist. Das Verhältnis aus beiden Werten nennt man Speichernutzungsgrad, kurz Nutzungsgrad, und synonym auch Füllgrad oder Speichereffizienz. Bei 100% sind alle physisch dem TextArray zugeordneten Speicherstellen auch durch ein logische Position vertreten.

Bei blockweiser Zuordnung von physischem Speicher zu einem TextArray ergibt sich, dass auch ein einzelnes Byte einen ganzen Block belegt, und den schlechtesten Nutzungsgrad ergibt. Zur Information sei gesagt, dass manche Zuordnungsverfahren für Sekundärspeicher versuchen, diesen schlechtesten Wert zumindest für viele Dateien zu vermeiden, indem sie bei kleinen Dateien zu besonderen Zuordnungsverfahren greifen, sodass sich beispielsweise mehrere Dateien einen Block teilen. Gerade in diesem Beispiel zeigt sich, dass der schlechteste Nutzungsgrad immer noch eintreten kann: bei einer einzelnen kleinen Datei.

Bezüglich der Eigenschaften des TextArray gibt man den durchschnittlichen Nutzungsgrad als Grenzwert für große Datenmengen an, theoretisch eine unendlich große Datenmenge. Tatsächlich sind heute typische Sekundärspeicher sehr groß und beinhalten Millionen von Speicherblöcken, sodass Prozentangaben des Nutzungsgrades bei voller Auslastung dieses Speichers hier äquivalent zum theoretischen Wert eines unendlichen Speichers sind.

(18) Separatoren im XEE

Wie in [ASTTA] beschrieben enthält die interne Textdarstellung des XEE Projektes erweiterte Trennzeichen, die nicht zum ursprünglichen Textinhalt des XML Dokumentes gehören. Diese Trennzeichen erleichtern die Mustererkennung im Textspeicher, wenn zwischen XML Auszeichnungen selbst kein Trennzeichen im Textinhalt erscheint, aber durch die XML Auszeichnung eine Trennung der angrenzenden Worte gegeben ist, die beachtet werden muss etwa bei generiertem XML aus einer Datenbank. Dort ist dann zwischen markierten Feldern eine Lücke zu setzen, die im reinen Textinhalt nicht existent sein muss.

Diese Separatoren verändern die logischen Position im Textinhalt des XML Dokumentes nicht, sie sollen in der Rekonstruktion aus dem AST/TA Speicher übersprungen werden. Die Beachtung dieses Umstands kann jedoch entfallen, wenn der AST Zugriffsbaum die Positionen von Textbereichen vermerkt unter Hinzunahme der Separatoren. Da sich der Zeichenwert des Separatorzeichens von allen anderen Textzeichen unterscheidet, kann der so angegebene Bereich dennoch eindeutig in seine ursprüngliche Form rekonstruiert werden. Die Länge der äußeren Darstellung eines Datenbereiches kann so jedoch geringer sein als die Differenz von Anfangs- und Endwert bezogen auf die interne Darstellung als Speicherung im TextArray mit Separatoren.

Insofern die Länge der äußeren Darstellung erkannt werden muss, ohne dass der Text im TextArray gelesen wird und die Separatorzeichen dabei erkannt werden, muss die Anzahl der Separatorzeichen in einem Bereich eben im Baum gespeichert werden. Die Speicherung solcher Werte kann dabei wahlweise im AST Zugriffsbaum oder im Speicherzuordnungsbaum des TextArray erfolgen. In der vorliegenden Implementation

wird sie im TextArray nicht beachtet - die Zugriffswerte auf das TextArray werden direkt umgesetzt und dabei Separatoren als normale Textzeichen betrachtet.

(19) Konkurrierende Zugriffe

Der gleichzeitige Zugriff mehrerer XEE Programme auf die gleiche Datenbasis auf dem Sekundärspeicher wird derzeit nicht mit Methoden unterstützt, die die gesicherte Ausführung garantieren, falls (quasi)parallel Veränderungs-Operationen am TextArray oder der AST Implementation auftreten.

Zuordnungsverfahren

Im dritten Teil wird schrittweise beschrieben wie ein logisch lineares TextArray auf Sekundärspeicher umgesetzt werden kann. Zur effizienten Unterstützung von Einfüge/Lösch-Operationen braucht es Lücken und Zuordnungsverfahren für nicht-angrenzende Blöcke des Sekundärspeichers. Die Zuordnung von Blöcken, Textabschnitten und Lücken werden verbunden in der Verwendung von Blockfüllständen, die Teil der Baumstrukturen des nächsten Abschnitts sind.

(20) Linear 1:1

Die naive Form der Zuordnung nutzt keine Umrechnung, man notiert nur die Anfangsposition des Speichers und findet jede beliebige physische Speicherstelle durch Additions-Operation der linearen logischen Position. Neben diesem Auffinden von Speicherstellen ist auch das Weitersetzen ohne Umrechnung möglich und der Nutzungsgrad geht gegen 100%, da es keine Lücken im TextArray gibt.

Während das Auffinden und Weitersetzen gar keine Zugriffe auf den Sekundärspeicher benötigt, wenn die Anfangsposition im Hauptspeicher registriert wurde, so ist das Einfügen/Löschen für diese Speicherform am ungünstigsten. Das Einfügen oder Löschen eines einzelnen Zeichens erfordert, dass alle nachfolgenden Zeichen physisch verschoben werden müssen, um die 1:1 Zuordnung von logischen zu physischen Adressen zu erhalten. Der schlechteste Fall ergibt sich bei großem Textblock und dem Einfügen/Löschen eines einzelnen Zeichens am Anfang, da alle Zeichen verschoben werden müssen, theoretisch also unendlich viele, nur begrenzt durch die tatsächliche Größe des physischen Speichers.

Diese Form nennt sich auch sequentielle Speicherung. Sie kam auf den ersten Speichermedien der Computergeschichte zum Einsatz und wird heute dort eingesetzt, wo nur geringe Mengen von Text verwaltet werden müssen. Beispiele sind ein TextArray im Hauptspeicher, Zuordnungen auf Smartcards, manche Speicherformate digitaler Audiorecorder und digitaler Photokameras arbeiten so. Auch das erste Format von Audio-CDs ist linear. Will man auf einer wiederbeschreibbaren CD ein Stück einfügen, müssen alle nachfolgenden Titel verschoben werden.

0 12 3..... 12 .....20.....34 = logische Position

Wilhelm Busch schrieb Max und Moritz

550 .....567.....570.....584 = Speicheraddresse

-insert (20, „ 1865“, 5):

0 12 3..... 12 .....20.....25.....39

Wilhelm Busch schrieb 1865 Max und Moritz

550 .....567.....570....575.....589 =verschoben

(21) Lücken

Statt bei jedem Einfügen alle nachfolgenden Zeichen verschieben zu müssen, lässt man im physischen Speicherblock mindestens eine Lücke. Beim Auffinden von logischen zu physischen Speicher verhält sich jeder Teil wie ein entsprechend kleinerer Teil mit linearer Zuordnung. Man benötigt jeweils den Anfang des physischen Speichers jedes Teils. Man prüft eine logische Position darauf, ob sie im unteren oder oberen Teil liegt. Bei letzterem addiert man zu dessen Anfangsspeicheradresse den logischen Abstand abzüglich der Anzahl der Zeichen, die im unteren Teil gespeichert sind.

Beim Weitersetzen muss darauf geachtet werden, ob eine Stelle an die Lücke grenzt und überspringt diese um auf die nächste physische Stelle zu einer logischen Position zu gelangen. Der Nutzungsgrad ergibt sich aus dem Verhältnis von Textlänge und Länge der Lücke.

Beim Einfügen/Löschen wird nur ein Bruchteil des Aufwands benötigt wie für die rein lineare Speicherung. Der Algorithmus prüft, ob die Einfüge/Löschposition sich an der Lücke befindet. Ist dies nicht der Fall, so werden alle Zeichen zwischen aktueller Position

0 12 3..... 12 .....13.....20.....34 = logische Position

Wilhelm Busch | | schrieb Max und Moritz

550 .....567.....588.....590.....604 = Speicheraddresse

-insert (20, „ 1865“, 5):

0 12 3..... 12 .....19.....20.....34 Rechter Rand der Lücke auf „20“

Wilhelm Busch schrieb | | Max und Moritz

550 .....567.....590.....604 =verschoben

0 12 3..... 12 .....19.....20.....25.....39

Wilhelm Busch schrieb | | 1865 Max und Moritz

Lücke verkürzen u. Text anfügen

550 .....567.....590.....604

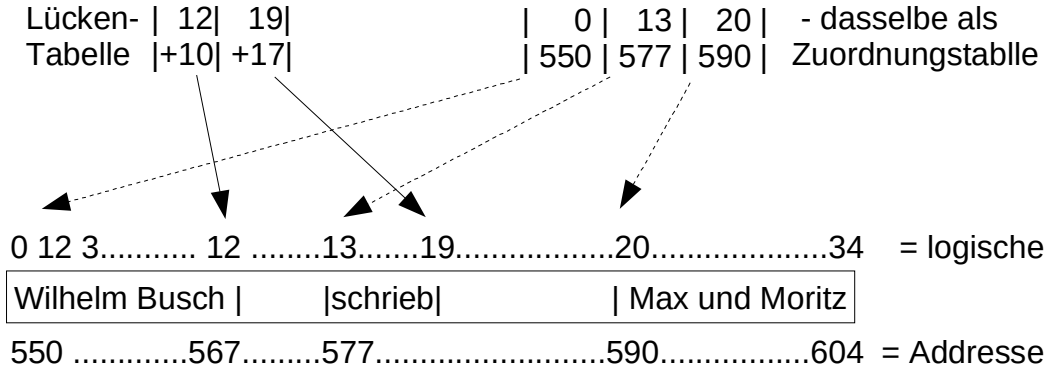
und der Lücke verschoben, sodass die Lücke an die Einfüge/Löschposition liegt. Beim Löschen vergrößert man nun die Lücke. Beim Einfügen verkürzt man sie entsprechend.

Ausgleich-Operationen werden nötig, wenn beim Löschen die Lücke zu groß wird und der Nutzungsgrad unter einen vorher definierten Wert fällt. Dann wird die Lücke verkürzt, indem der obere Textteil von der Lücke zum Ende verschoben wird. Wenn dagegen beim Einfügen die Lücke auf Null sinkt und weitere Zeichen eingefügt werden müssen, so wird eine neue Lücke geschaffen, indem der obere Block verschoben wird.

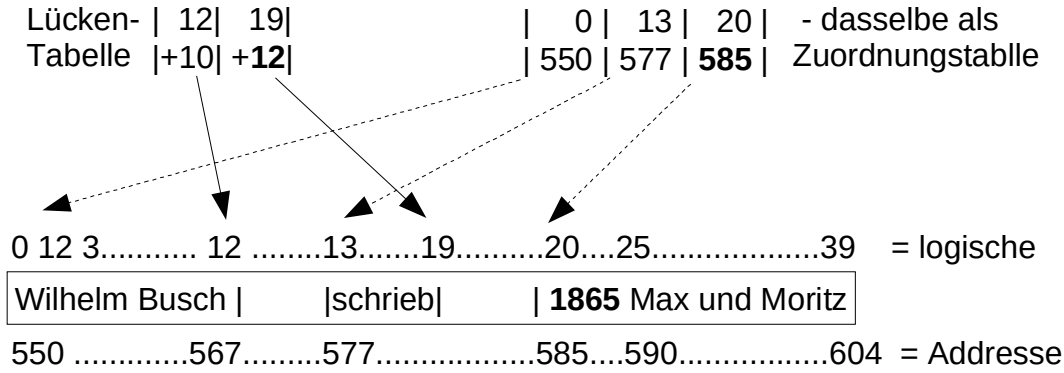
Schon eine einzelne Lücke ermöglicht es, dass das Versetzen der Lücke im Durchschnitt ein Drittel des Textspeicher physisch verschiebt. Dieses Zuordnungsverfahren ist besonders dort günstig, wo Einfügen/Löschen mehrfach hintereinander an gleicher logischer Stelle erfolgt, da dann kein Versetzen der physischen Lücke erforderlich wird.

Dieses Zuordnungsverfahren ist aus dem Emacs Texteditor bekannt, da gerade beim Editieren von Text die Zeichen sehr oft einzeln/gruppiert eingetippt/gelöscht werden. Beim Emacs wird es im Hauptspeicher eingesetzt, da Emacs in einem Zeitrahmen der Computergeschichte entstand, wo das Verschieben auch Hunderter Kilobyte soviel Zeit verbrauchte, dass es für den Menschen spürbar war.

Das Lückenverfahren kann auch für mehrere Lücken verallgemeinert werden, die auf dem Sekundärspeicher in einer Zuordnungstabelle vermerkt werden. Ist es eine lineare Tabelle mit Paaren aus physischer Anfangsposition eines Teils und zugehöriger logischer Position, so kann das Auffinden des physischen Teils durch binäre Suche schnell erfolgen. Auf



-einfügen der Jahreszahl:



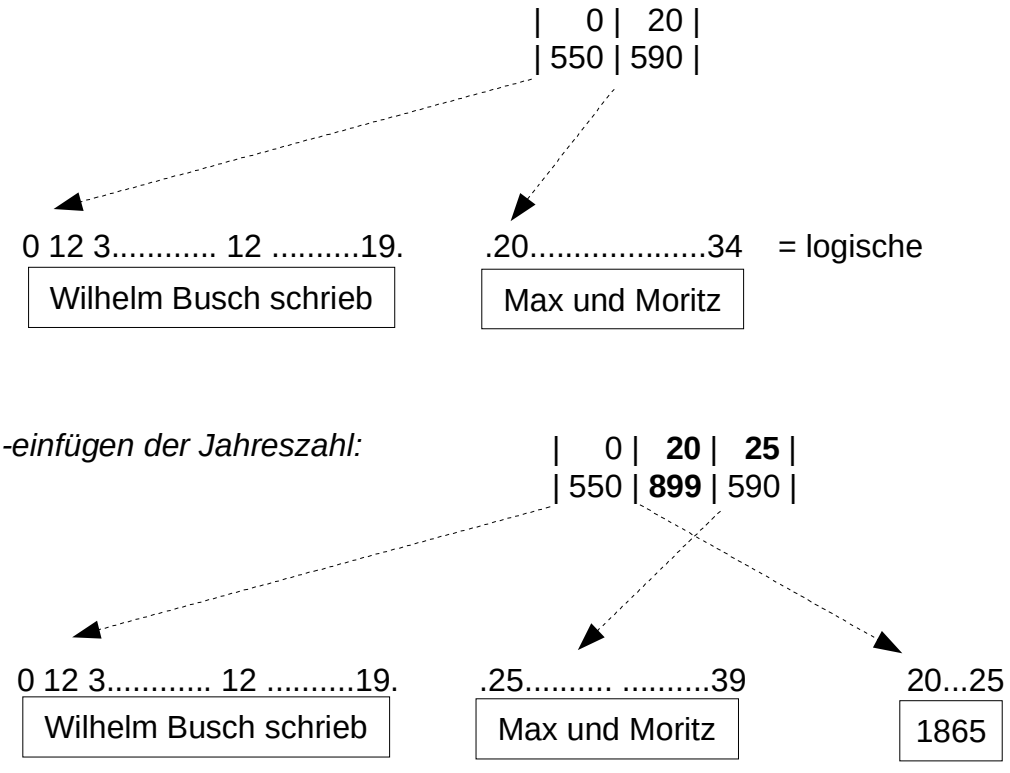
Sekundärspeicher wird jedoch eine blockorientierte Form bevorzugt, die weiter unten angesprochen wird.

(22) Blockzuordnungstabelle

Sekundärspeicher wird fast immer in Blöcken angesprochen. Bei der Zuordnung zu einem Textspeicher müssen diese Blöcke nicht physisch aufsteigend liegen, wenn man eine Zuordnungstabelle benutzt. Dieses Verfahren ist im FAT-Dateisystem (FAT = File Allocation Table) weit verbreitet. Es speichert eine Zuordnungstabelle am Anfang des Speichers, für jeden Block des Speichers genau ein Eintrag.

Wenn das Weitersetzen einer Textposition an das Ende eines Blocks kommt, so findet sich in der FAT Zuordnungstabelle im Eintrag die Nummer des nächsten Blocks. Der nächste Speicherblock muss so nicht mehr physisch nachfolgend angeordnet sein. Das Auffinden einer Position ist dagegen deutlich langsamer, da im FAT die Blockzuordnung eine Liste bildet, die durchsucht werden muss. Das Einfügen ganzer Blöcke erlaubt hier, dass ein neuer Block in die Liste eingehängt wird, eine klassische und schnelle Operation der Programmierung. Ebenso kann beim Löschen der Block heraus gelöst werden, ohne dass die nachfolgenden physischen Blöcke verschoben werden.

Gerade die Listenform der FAT Zuordnungstabelle ist ungünstig für große Dateien, da das Auffinden linear ansteigt mit der Größe der Speicherzuordnung. Eine kompakte Tabelle ist hier günstiger, wie sie auch in vielen anderen Dateisystemen zu finden ist. Dabei arbeiten aber Dateisysteme auch dort auf Blockebene, die eine Umordnung logischer Blöcke zu physischen vornehmen. Das Löschen einzelner Zeichen erfordert, dass nicht nur der Speicher im Block des Einfügens/Löschens verschoben wird, sondern auch die aller



nachfolgenden physischen Blöcke. Verwendet man nur eine Blockumordnung, sind die gespeicherten Zeichen eben weiterhin lückenlos angeordnet.

(23) Blockfüllstände

Man kombiniert nun beide Teile, sowohl eine Umordnung der Speicherblöcke als auch das Einbringen von Lücken. Jeder Block bekommt seine eigene Lücke, indem jedem zugeordneten Speicherblock ein Füllstand zugeordnet wird. Im einfachsten Fall, befindet sich der genutzte Teil am Anfang des Speicherblocks, die Lücke am Ende. Bei einer einfachen Blockumordnung musste nur der Zielblock gespeichert werden, die logische Position vor der Umordnung ergab sich implizit, da die Blöcke gleichgroß sind und logisch lückenlos. Mit Lücken ist der Nutzungsgrad jedes Blocks verschieden, sodass die logische Position explizit verzeichnet wird.

Das Auffinden einer physischen Position kann bei kompakter Speicherung durch binäre Suche in den logischen Positionen erfolgen. Das Weitersetzen ist ebenfalls einfach: man prüft auf das Ende des genutzten Bereichs im Block und fragt in der Blockzuordnungstabellen den nachfolgenden Eintrag ab, der den Anfang des nächsten physischen Blocks anzeigt. Das Einfügen/Löschen auch einzelner Zeichen ist schnell, da die Lücke entsprechend vergrößert oder verkleinert wird. Werden beim Einfügen mehr Speicherstellen gebraucht als im aktuellen Block in der Lücke verfügbar sind, so kann ein leerer Block in die Umordnungstabelle eingefügt werden.

Komplex wird die Implementation durch zwei Ereignisse. Zum einen kann durch Löschen die Lücke in jedem Block sehr groß werden, sodass der Nutzungsgrad des Textspeicher gering ist. Dem begegnet man mit Ausgleichen der Füllstände. Zum anderen kann durch Einfügen die Umordnungstabelle sehr groß werden, sodass das Einfügen/Löschen neuer Blöcke die gleichen Verzögerungen aufweist wie das Einfügen einzelner Zeichen in einen großen Textspeicher. Dem begegnet man mit Mehrwegebäumen. Beides wird folgend beschrieben.

Mehrwegebaum

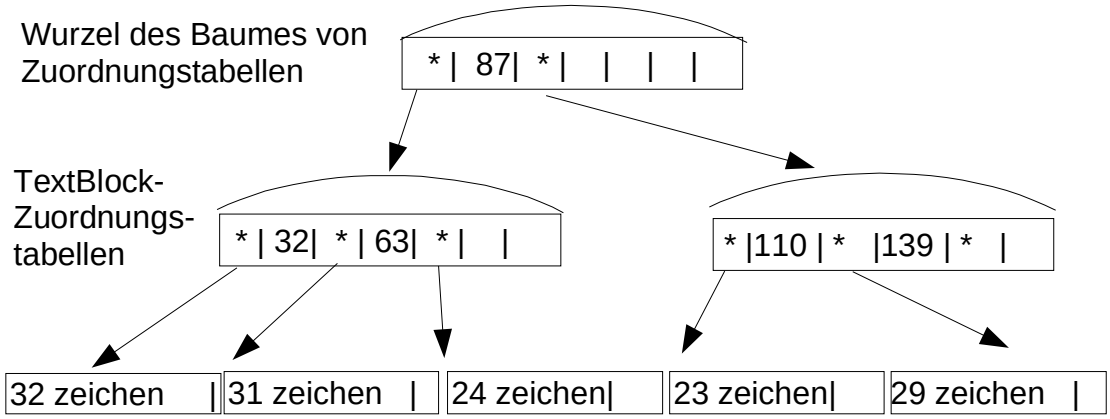
Im letzten Teil führen wir die Teile zusammen. Die Textabschnitte in den Blöcken des Sekundärspeichers werden durch eine mehrstufigen Baumstruktur verwaltet, die auch selbst Füllstände verwenden. Zur Erhaltung eines mindestens Nutzungsgrades werden Ausgleichs-Operationen auf den verschiedenen Ebenen des Mehrwegebaumes verwendet.

(24) Aufbau

Um eine binäre Suche nach der Position zu ermöglichen, arbeitet man in Zuordnungstabellen mit einem kompakten Format. Wenn neue Datenblöcke dem TextArray zugeordnet werden oder andere geleert wurden und entfernt werden sollen, so muss dabei das lückenlose Format erhalten bleiben. Der Aufwand für das Einfügen/Löschen eines Eintrags steigt nun linear mit der Größe. Sie ist jedoch um einen Teiler geringer, der dem mittleren Nutzungsgrad entspricht.

Statt einer einzelnen langen Zuordnungstabelle kann man eine weitere Zuordnungstabelle nächsthöherer Ordnung schaffen. Die Zuordnungstabelle wird dazu selbst auf Blöcke

zugeschnitten, die nur teilweise bis zu einem Mindestgrad gefüllt sind. Diese teilweise gefüllten Blöcke werden dann referenziert und die logische Position des Anfangs der Zuordnungstabelle gespeichert. Diese Schachtelung lässt sich rekursive wiederholen und ergibt so einen Zuordnungsbaum statt einer linearen Tabelle.



Die Beschränkung der Teile der Zuordnungstabellen auf Blockgrößen des Sekundärspeichers ergibt eine schnelle und einfache Verwaltung. Jeder Block jeder Ebene ist dabei wiederum nur teilweise gefüllt, sodass bei Aufnahme neuer Blöcke nur geringe Verschiebe-Operationen nötig werden. Mit einem Zuordnungsbaum können sehr große Blockmengen verwaltet werden - die Anzahl ergibt sich aus der Anzahl der Einträge je Zuordnungstabelle potenziert mit der Anzahl der Ebenen.

(25) Auffinden

In der obersten Zuordnungstabelle wird nach dem entsprechenden Eintrag gesucht, in den die logische Position passt. Die angegebene Blockindex verweist dann auf eine weitere Zuordnungstabelle, die nur Zuordnungen von Teilen des Bereichs der logischen Positionen enthält, die dem Eintrag in der übergeordneten Tabelle entsprechen. Auf diese Art verfeinert man in jeder Ebene den Bereich der logischen Positionen, die unter einem Eintrag zu finden sind.

Ein Mehrwegebaum speichert in jeder Ebene mehrere Verweise auf untergeordnete Zuordnungstabellen. Dadurch ist ein Mehrwegebaum auch bei großer Zahl zugeordneter Textblöcke sehr flach. Ein Baum mit maximal 32 Einträgen je Zuordnungsblock und 4 Ebenen kann $32^4 = 2^{20}$ Textblöcke verwalten, bei maximal 1 KByte je Textblock also maximal 1 GigaByte. Zum Auffinden einer Textstelle bei 4 Ebenen genügen jeweils Zugriffe auf je einen Block jeder Ebene des Zuordnungsbaumes, sodass der fünfte Zugriff den Block des Textabschnitts betrifft. Dies ist sehr viel geringer als das Durchsuchen einer langen Zuordnungstabelle mit 2^{20} Einträgen.

(26) Weitersetzen

Das Weitersetzen von einem Datenblock in den nächsten beinhaltet, den nächsten Eintrag in dessen Zuordnungstabelle zu holen. Langt man an das Ende der Zuordnungstabelle, so muss der erste Eintrag der nächsten Zuordnungstabelle geholt werden. Diese Zuordnungstabelle zu finden, gibt es üblich zwei Wege.

Der implizite Weg besteht darin, zur übergeordneten Zuordnungstabelle der gegenwärtigen zu gehen, den nächsten Eintrag zu holen, diesen wieder als Zuordnungstabelle zu nehmen und den ersten Eintrag zu entnehmen. Erstreckt sich der Zuordnungsbaum über mehrere Ebenen, besteht der ungünstigste Fall immer darin, bis zur Wurzel hoch zu gehen, um dort den nächsten Eintrag zu holen, und von dort wieder bis zu den Blättern des Baumes hinab zu dessen ersten Eintrag zu gehen. Da ein Mehrwegebaum nicht sehr tief ist, ist das meist schnell genug.

Der explizite Weg besteht darin, die Zuordnungstabellen untereinander noch zu verbinden, also noch einen Eintrag mitzugeben auf die direkt nächstfolgenden Zuordnungstabelle gleicher Ebene. Dadurch reduziert sich der schlechteste Fall auf einen Schritt. Bei Einfügungen neuer Blöcke für Zuordnungstabellen müssen dabei auch diese Verweiseinträge mitverwaltet werden. Dies erfordert mindestens zwei weitere Blöcke auf dem Sekundärspeicher zu verändern, für die Verweise im vorhergehenden und nachfolgend zugeordneten Block. Diese schreibenden Zugriffe zur Verwaltung der Referenzen kommen jedoch sehr viel seltener vor als das Auffinden und Weitersetzen von Positionen zum logisch angrenzend zugeordneten Block.

(27) Löschen / Zusammenfügen

Beim Löschen von Zeichen in einem Block wird dessen Füllstand geringer und der freie Bereich in dessen Lücke größer. Wenn der Block dabei nicht leer wird, kann er nicht freigegeben werden. Im schlechtesten Fall belegt dann ein einzelnes Zeichen einen ganzen Block. Dieser Zustand kann im schlechtesten Fall auf alle Blöcke zutreffen, die einem TextArray zugeordnet sind.

Wenn der Block nicht ganz entleert ist, kann man nun prüfen, wie der Füllstand des vorhergehenden (oder auch nachfolgenden) Blocks ist. Ist die Lücke im nebenliegenden Block größer oder gleich der noch verbliebenen Bytes im aktuellen Block, so kann man die verbliebenen Bytes in den nebenliegenden Block verschieben. Logisch sind die Bytes ja schon angrenzend, es wird hier die physische Lücke zwischen ihnen entfernt, die TextArray Zeichen werden zusammengefügt, und damit werden gleichzeitig auch die Lücken beider Blöcke zusammengefügt. Ist die Summe der Textzeichen beider Blöcke kleiner als die Blockgröße, so ist die Lücke größer als ein Block, und folglich ein Block komplett leer und kann freigegeben werden, also aus der Zuordnungstabelle entfernt werden.

Durch Zusammenfügen zweier angrenzender Blöcke erhöht man den schlechtesten Fall für den Nutzungsgrad – bei zwei Blöcken gilt, dass die Summe der genutzten Bytes größer als ein Block ist, andernfalls würde einer der beiden aus der Zuordnung entfernt, wodurch der Nutzungsgrad des verbliebenen einen Blocks das Ergebnis ist, der mithin doppelt so hoch ist als das Ergebnis, wenn beide Blöcke in der Zuordnung verblieben Es ergibt sich ein Nutzungsgrad von mindestens 50% nach dem Zusammenfügen

(28) Einfügen / Ausgleichen

Der eben angedeutete wiederholte Test auf Zusammenfügen von Blöcken ist nicht nötig, um einen 50% Nutzungsgrad über den gesamten Textspeicher zu erreichen. Es genügt ein einzelner Test auf Zusammenfügen mit dem vorherigen Block.

Bei Betrachtung von zwei Blöcken beachtet man auch das Ausgleichen beim Einfügen. Wenn ein Einfügen in einen Block nicht vollständig in dessen Lücke passt, so wird ein

leerer Block in die Zuordnungstabelle aufgenommen, der dem aktuellen folgt. So wird die Lücke vergrößert und kann die einzufügenden Bytes aufnehmen.

Dabei lässt man jedoch den ersten Block nicht vollständig gefüllt und gibt dem nachfolgenden die überzähligen Bytes. Dies könnte zur Folge haben, dass die nächste Einfüge-Operation auf den vollen Block erneut einen leeren Block einfügen würde. Stattdessen teilt man die Bytes gleichmäßig auf die zugeordneten Blöcke auf. Es ergibt sich folglich, dass jeder Block mindestens bis zur halben Blöckgröße gefüllt ist.

Wenn man zufällig Text in das TextArray einfügt oder aus diesem löscht, so ergibt sich ein besserer Durchschnitt als 50%, das statistische Mittel findet sich bei zwei drittel. Die 50% sind die unterste Grenze im schlechtesten Fall, wenn man in jedem Einfügen und Löschen prüft, und sich ergibt, dass die Summe der Bytes mit dem vorhergehenden (oder nachfolgenden) Block gerade um ein Byte größer ist als in einen Block passen.

Der schlechteste Nutzungsgrad von 50% ist in manchen Anwendungsfällen nicht gut genug. Statt mit einem angrenzenden Block den Füllstand auszugleichen, kann man dazu übergehen, mit den beiden angrenzenden Blöcken auszugleichen. Man prüft hier, ob die Summe der Textbytes in den drei Blöcken größer als zwei Blöckgrößen ist.

Andernfalls fügt man die verbliebenen Textbytes der drei Blöcke in zwei Blöcken zusammen und entfernt den leeren Block aus der Zuordnungstabelle des TextArray. Gleichfalls wird beim Einfügen von Daten ein neu eingeordneter Block mit zwei nebenliegenden Blöcken ausgeglichen auf gleichen Füllstand. Dadurch ist hier der schlechteste Fall schon 66%, der typische Fall liegt wieder ein Drittel darüber.

(29) Mehrstufiges Ausgleichen

Das Einfügen/Löschen von Bytes innerhalb eines Blocks kann durch Verschieben der Bytes in diesem Datenblock erfolgen und anschließendes Anpassen des Füllstandes für diesen Datenblock mit dem Textabschnitt. Hier werden wiederum die Methoden des Ausgleichens eingsetzt, wie in den Abschnitten zum Einfügen und Löschen beschrieben, wenn ein Mindestnutzungsgrad nicht mehr gegeben ist. In der Folge werden auch Bytes in benachbarten Datenblöcken verschoben und deren Füllstand angepasst.

Das Ausgleichen kann beim Löschen einige Datenblöcke freigeben oder zur Speicherung der Daten beim Einfügen weiterer Datenblöcke benötigen. Diese müssen in die Zuordnungstabelle eingefügt oder entfernt werden. Dies wird nachvollzogen durch Einfügen/Löschen von Einträgen in der Zuordnungstabelle. Die Zuordnungstabelle besitzt einen Füllstand, deren Elementgröße der eines Eintrags entspricht. Hier wird die Methode des Ausgleichens wiederum angewendet, etwa wenn beim Löschen in einer Zuordnungstabelle der Zustand eines Unterlaufens eines Mindestnutzungsgrades eintritt. Dann werden nachfolgende Einträge aus der logisch nachfolgenden Zuordnungstabelle gleicher Ebene hinzugefügt oder die verbliebenen Einträge in die nebenliegende Zuordnungstabellen verschoben, sodass eine (oder mehrere) Zuordnungstabellen leer werden. Dieses führt rekursiv wiederum dazu, dass Einträge in der nächsthöheren Zuordnungstabelle entfernt werden.

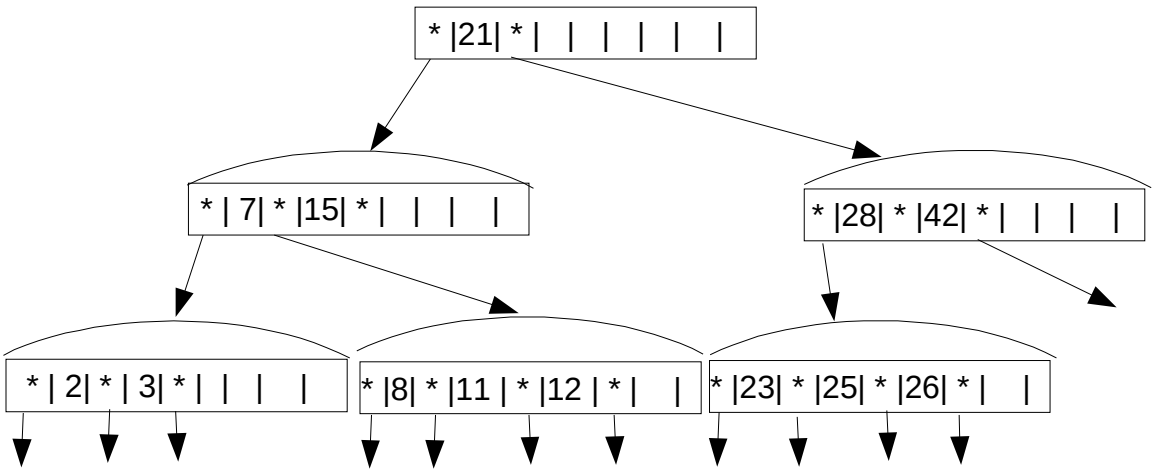
Grundsätzlich kann für jede Ebene im Mehrwegebaum ein anderes Verfahren zum Ausgleichen eingesetzt werden – etwa das Ausgleichen unter zwei oder drei Blöcken wie oben beschrieben. Der besondere Fall bei einem Mehrwegebaum besteht darin, wie hoch er ist, wieviele Ebenen also zur Speicherung der verschachtelten Zuordnungstabellen

benötigt werden. Beim Einfügen kann der Fall eintreten, dass alle Einträge in allen Ebenen gefüllt sind, aber weitere Einträge benötigt werden. In realen Systemen ist es durchaus möglich, die maximale Höhe des Mehrwegebaumes vorher festzulegen in Abhängigkeit von der Größe des Sekundärspeichers, sodass eher keine Datenblöcke mehr verfügbar als keine Einträge mehr möglich sind.

Legt man die Höhe des Mehrwegebaumes im Vorhinein fest, so wird für eine kleine Anzahl von Datenblöcken der Fall eintreten, dass fast alle Ebenen eine Zuordnungstabelle enthalten mit genau einem Eintrag. Bei Speicherung eines einzelnen Byte im TextArray wird dennoch ein Zuordnungstabelle für jede Ebene des Baumes gebraucht. Dieser Zustand kann vermieden werden bei Einsatz von Algorithmen zu balancierten Bäumen mit dynamischer Höhe, der auch den administrativen Prozess der vorherigen Festlegung der (damit maximalen) Höhe des Baumes erübrigt. Algorithmen zu höhenbalancierten Bäumen (B-Bäume) sind bei Sekundärspeicher Mehrwegebäume, bedingt durch die Blockorientierung des Sekundärspeichers.

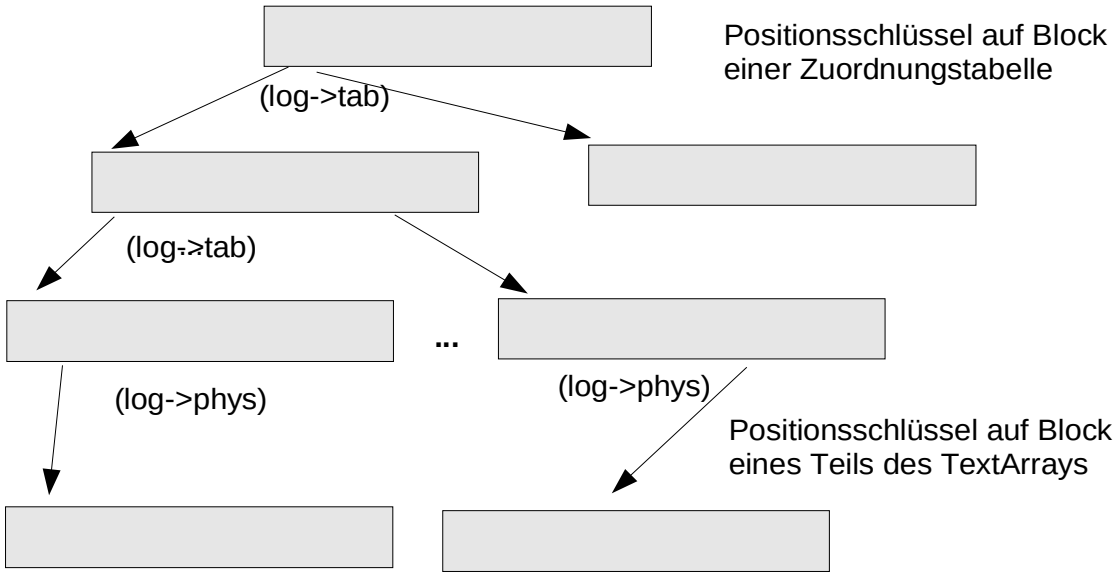
3. Implementation

Die verwendete Form des höhenbalancierten Mehrwegebaumes ist die bekannte Form eines Positions-B-Baum. Hierbei wird der Positionsindex der Elemente des TextArrays als Schlüsselwerte verwendet. Die Speicherung erfolgt auf Sekundärspeicher [EXODUS].



(30) B⁺-Bäume und B-Bäume*

In der Literatur finden sich Bezeichnungen für typische B-Bäume. Ein B*-Baum etwa ist minimal zu 2/3 gefüllt, mit entsprechend aufwändigen Ausgleichs-Operationen. Ein B⁺-Baum referenziert Datenblöcke nur in den Endknoten. Wir betrachten in dieser Arbeit nur eben letztere, da es das Auffinden von Positionen im TextArray erleichtert.



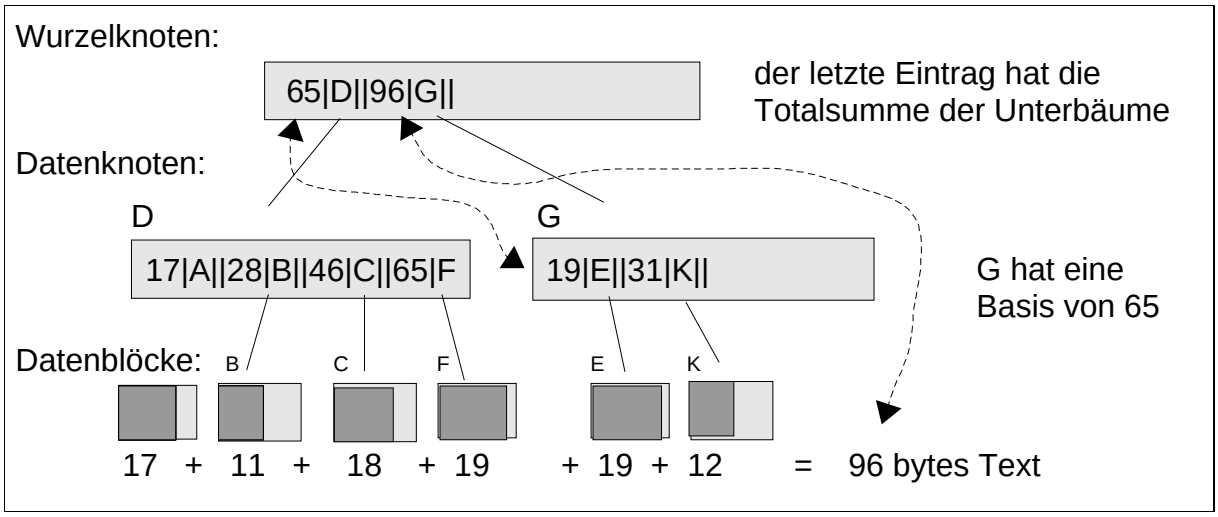
In der Literatur findet sich, dass B⁺-Bäume regelmäßig in den Endknoten querverlinkt sind, um schnelles Weiterschalten zu ermöglichen. Diese Form wurde oben bereits beschrieben bei den Mehrwegbäumen. Die vorliegende Implementation beinhaltet nur einen einfachen Positions-B⁺-Baum ohne 2/3-Ausgleich wie im B*-Baum

(31) Differentielle Positionsschlüssel

Die naheliegendste Form der Speicherung von Textteilen ist die Aufzeichnung von Paaren aus Blockindex und der Anzahl der Bytes, die in dem angegebenen Datenblock genutzt werden. Mehrere dieser Paare werden zusammengefasst bis auf die Größe eines Knotenblocks, die so zusammen auf dem Sekundärspeicher liegen. Die Summe der Bytes, die durch alle Paare gehalten wird, ist die Anzahl der Bytes, die dem Knotenblock zugeordnet werden. Höhere Knoten im Zugriffsbaum speichern dann Paare aus Blockindex des unterliegenden Knotenblocks und der darin enthaltenen Bytesumme.

Der Nachteil dieser Speicherung ist jedoch, dass es das Auffinden einer Textposition erfordert, schrittweise vom Anfang eines Knotenblocks zu beginnen und mitzuaddieren, bis man soviele Bytes übergangen hat, dass der nächste Index schon über der Textposition liegt, die gefunden werden soll. Statt der Addition während der Suche speichert man besser die voraddierte Bytesumme aus allen vorhergehenden Paaren und der im Blockindex enthaltenen Bytesumme, genannt die Totalsumme. So kann in der Implementation eine binäre Suche auf die Liste der Paare anwenden, die somit jenen Eintrag findet, dessen Totalsumme gerade über der gesuchten Textposition liegt.

Diese Form ist nicht identisch mit einer anderen naheliegenden Form der Speicherung, bei der jeweils Paare aus Blockindex und der Textposition gebildet werden, bei der die Textposition des Paars dem Byte am Anfang des zugeordneten Datenblocks entspricht. Wird hier eine längenverändernde Operation im TextArray ausgeführt, so müssten dann alle nachfolgenden Textpositionen modifiziert werden, und zwar nicht nur im Knotenblock der Einfüge-Operation, sondern auch aller in der logischen Ordnung nachfolgenden Knotenblöcke.



In der implementierten Form der Totalsummen beginnt jeder Knotenblock mit enthaltenen Paaren bei Null und speichert in jedem Paar die relative Bytesumme der Differenz zur Position der vorhergehenden Knoten, also ein differentielle Positionsschlüssel. Nur in der Baumwurzel entspricht dies dann den logischen Textpositionen - bei tieferen Knoten ist nur jener Anteil gespeichert, der nicht in der Bytesumme der übergangen Textknoten bis zum Eintrag im Vaterknoten liegt, der auf die aktuelle Zuordnungstabelle zeigt. Dadurch bleiben längenverändernde Operationen beschränkt auf die Knotenblöcke, die auf dem Pfad der Baumwurzel zum Knotenblock mit dem Datenblockindex liegen. Eine

Aufsummierung im Vaterknoten verschiebt die logische Basis aller später liegenden Kindsknoten, die nicht angefasst werden müssen, da diese nur jeweils den relativen Abstand zu der logischen Basis enthalten, die in ihrem Eintrag im Vaterknoten gespeichert ist.

Der Nachteil ist, dass bei mehrstufigen Zuordnungstabellen die Endknoten der Zuordnungsstruktur nicht die logische Textposition enthalten, die beim Abspeichern der Daten des TextArray verwendet wurde. Die eigentliche Textposition eines Eintrags findet sich erst durch Addition der Bytesummen jeder Ebene bis zur Wurzel des Zuordnungsbaumes, bei dem der Bytesummen Eintrag der logischen Textposition entspricht. Es zeigte sich in der Implementation als günstig, für bestimmte Operationen den Zugriffspfad durch den Zuordnungsbaum zu vermerken, um schnell auf die differentiellen Positionen in den Zwischenknoten zugreifen zu können.

(32) Einbettung in das XEE Projekt

Das XEE Projekt besitzt eigene Hilfsroutinen zur Verwaltung des Sekundärspeichers. Die verwendete Blockgröße des TextArray ist dabei einstellbar und nicht durch den tatsächlichen physischen Speicher vorgegeben. Die Blöcke des Sekundärspeichers können verschiedenen Teilen des XEE Projektes zugeordnet werden, das TextArray ist dabei nur ein Teil, andere Teile speichern beispielsweise den Auszeichnungsbaum oder Wort Indexe des Projektes.

Die vorliegenden Variante der Implementation des TextArray verwendet die weithin bekannten Algorithmen des positional B*-Tree mit differentiellen Positionsschlüsseln. Die Reimplementation ermöglicht spezielle Erweiterungen des Baumes, etwa die Vorhaltung der Separatorsummen im Zugriffsbaum des TextArray. Diese sind derzeit nicht implementiert, und Teil des Abschnitts dieser Arbeit zu zukünftigen Erweiterungen.

(33) Abbildung der Operationen

In der Implementation werden alle Veränderungs-Operationen über Abschnitten des TextArrays auf eine einzige aufzurufende Operation abgebildet. Die Abschnitte werden beschrieben als Textspanne im Textinhalt ab einer logische Anfangsposition und einer Anzahl nachfolgender logischer Stellen, der Länge der Textspanne. Bei Veränderungs-Operationen wird der neue Text und dessen Länge mitgegeben. Unterscheiden sich die Längen von alter Textspanne zu neuem Ersetzungstext so verändert sich entsprechend die Gesamtlänge des TextArray um deren Differenz.

In der folgenden Tabelle wird der lesenden Funktion "read" die Textspanne angegeben als Paar von Anfangsposition P des Textes im TextArray und dessen Länge N. Die schreibende Funktion "update" erhält ein zweites Paar, einer Referenz T auf das neue Textstück und dessen Länge L. Das Löschen und Einfügen wird realisiert indem in je einem der Paare die Länge auf Null gesetzt ist.

typische logische Operation	***Abbildung für den Aufruf***
Lesen von Textinhalt ab Position P aufsteigend über Länge N	read (P, N)
Lesen des Textes zwischen Position A und Position B	Anfang ist die kleinere Position, Länge der Abstandes beider Positionen: read (min(A,B), max(A,B)-min(A,B))
Einfügen von Text T der Länge L an Position P im Textinhalt	die Textspanne im Textinhalt hat die Länge Null, beginnend bei P. Durch die neue Länge L erweitert sich die Gesamtlänge des TextArray um L update((P,0) -> (T,L))
Löschen von Textinhalt ab Position P aufsteigend über Länge N	die Angegeben Textspanne wird durch eine leeren Text ersetzt. update ((P,N) -> (0,0))
Löschen von Textinhalt zwischen Position A und Position B	anzugebende Werte P und L für die Textspanne können wieder durch Differenz gebildet werden P := min(A,B) N := max(A,B) – min (A,B) update ((P,N) -> (0,0))
Ersetzen des Textinhaltes zwischen Position A und Position B durch einen Text T der Länge L	wenn bekannt ist, dass A < B, so ist Angabe der Textspanne einfacher update ((A,B-A) -> (T,L)) die Gesamtlänge des TextArray kann verändert werden, wenn B-A <> L

Die Umsetzung ab dem funktionellen Aufruf geschieht durch Kombination der Grundoperationen im Zuordnungsbaum des TextArray. Zur logischen Position P muss dessen physische Position (Q,R) gefunden werden, einem Paar aus Blockindex auf dem Sekundärspeicher und dem Abstand des Textelementes vom Anfang des Blocks – die "Auffinden"-Operation kann über mehrere Ebenen im Zuordnungsbaum gehen. Wenn die Textspanne nicht komplett in dem Block Q der logischen Anfangsposition P liegt, dessen Füllgrad F also nicht reicht (R+L > F), so muss die Lese/Schreib-Position auf einen neuen Block weitergesetzt werden – nach der "Weitersetzen"-Operation kann der Pfad durch die Ebenen der Zuordnungstabellen des Zuordnungsbaumes sich komplett geändert haben.

Beim "read" werden die zugeordneten Blöcke des Sekundärspeichers nur geholt, beim "update" werde sie auch zurückgeschrieben. Die Implementation des "upate" arbeitet immer in zwei Phasen, es überschreibt zuerst die schon zugeordneten Blöcke mit dem neuen Text. Wenn Ersetzungstext übrig bleibt, der nicht in die schon zugeordneten Blöcke passt, so werden neue Blöcke eingefügt und ihnen ein gleichmäßiger Füllgrad gegeben. Wenn der Ersetzungstext leer wurde, aus der alte Textspanne aber noch weitere Elemente in den Blöcken auf dem Sekundärspeicher enthalten sind, so werden entsprechend Blöcke gelöscht und mit dem jeweils letzten Block der Füllstand ausgeglichen. Der Block am Übergang der beiden Phasen ("overwrite-point") wird jeweils

speziell beachtet, ähnlich wie der letzte Block, um einen mindesten Füllgrad für alle zugeordneten Blöcke sicher zu stellen.

Als Beispiel für enthaltene Grundoperationen, hier die Angabe der Lesen-Operation

read (P,N) mit N > 0	(1) "Auffinden" der physischen Position (2) "Holen" des Speicherblocks (3) "Kopieren" der Textelemente (4) wenn N Textelemente kopiert: Ende (5) sonst, "Weitersetzen" und weiter mit (2)

Die Veränderungs-Operationen sind aufwendiger, da jeweils Ausgleichs-Operationen auftreten, je am Block des "overwrite-point" und am letzten Block, der der ursprünglichen Textspanne zugeordnet ist ("end-point"). Findet sich in der zweiten Phase, dass Blöcke aus der Zuordnung zum Textinhalt entfernt werden müssen, oder dazugefügt werden, so führt dies zu Veränderungs-Operationen im Zuordnungsbaum. Dieser arbeitet ebenfalls mit Füllständen, deren Grundelement hier Paare aus Blockindex und Füllstand sind (der Füllstand ergibt sich hier speziell aus den Totalsummen der zugeordneten Blöcke). Diese Veränderungen können sich rekursiv durch die Ebenen des Mehrwegebaumes fortsetzen, entlang des Pfades zur physischen Position. Diese Methodik wurde schon in [EXODUS] beschrieben, mit Pfaden zum "overwrite-point" und "end-point" und dem Ausgleichen der Ebenen. Die Implementation enthält weiterhin lange Fallunterscheidungen für das Ausgleichen in den Blöcken am "overwrite-point" und "end-point", die ja auch den gleichen physischen Block betreffen können, oder in der logischen Zuordnung direkt angrenzend sind. Diese Fälle folgen jeweils den beschriebenen Methoden, je für Blöcke mit Textinhalt und den Blöcken des Zuordnungsbaumes.

(34) Messungen - Einlesen von Dokumenten

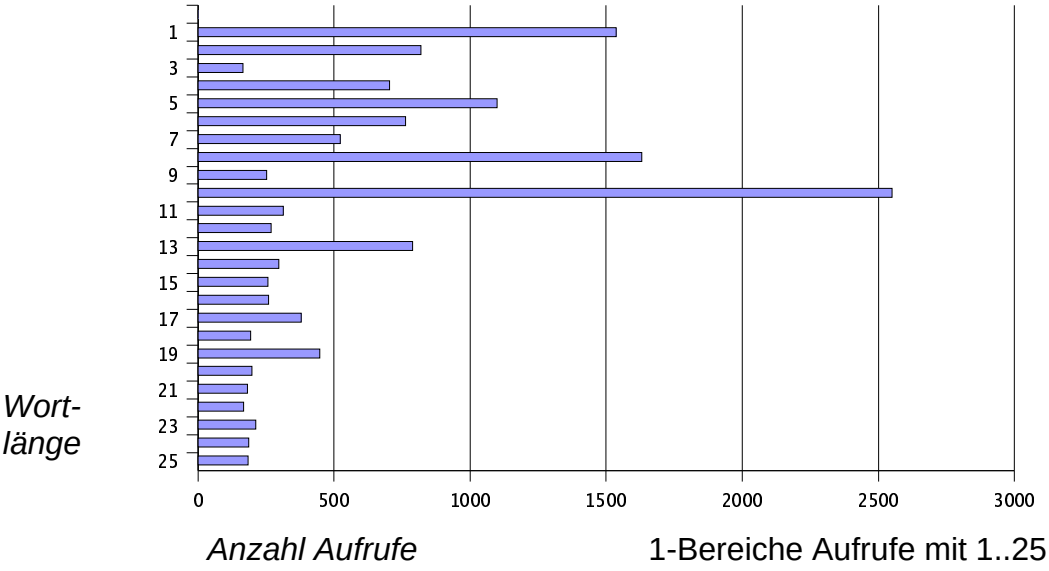
Derzeit ist im XEE Projekt das Einlesen einer XML Datei in die Datenbank implementiert. Bei den Tests werden dabei synthetische Dokumente verwendet, die von xmlgen generiert wurden. Dieses Programm soll eine typische XML Struktur nachahmen für beliebige Größen. Die Verteilung der schreibenden Aufrufe auf das TextArray, die beim Einlesen der Datei entstehen, können mitgeschnitten werden und der daraus resultierende typische Nutzungsgrad in den Blöcken des Sekundärspeichers untersucht werden.

Die nachfolgenden Tabellen beziehen sich durchgehend auf das Verhalten der Implementation bezüglich der 0.02 Datei – dies ist eine 2 Megabyte große XML Datei, die von xmlgen erzeugt wurde. Das xmlgen Werkzeug ist dabei einem anderen Projekt entnommen, und erstellt synthetische Dokumente, die dem Aufbau und Inhalt einer typischen XML Textdatei möglichst nahe kommen sollen. Die tatsächliche Form von XML Dateien kann bekanntlich sehr verschieden sein. Für Gewinnung von Erkenntnissen zur TextArray Implementation aus Messungen wird die Struktur als hinreichend typisch angenommen, um als Beispiel Dokument Anwendung finden zu können.

Die 0.02 Datei ist zum derzeitigen Stand die größte Datei, die in regelmäßigen Funktionstests aufgerufen wird, um die Funktionalität des XEE im Laufe der Fortentwicklung sicherzustellen. Die dargestellten internen Strukturen werden hier als

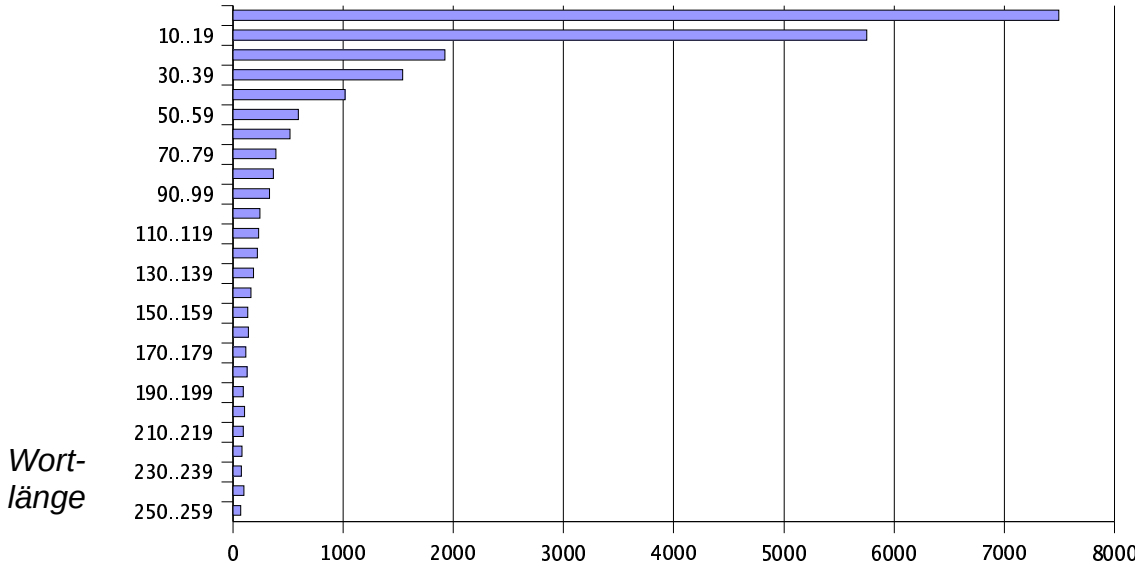
fehlerfrei angenommen, und das Verhalten der Implementation als bezeichnend auch für späteren Verwendungen des XEE gesehen, einschließlich der enthalten TextArray Daten. Mit den gemessenen Werten soll auf das erwartbare Verhalten im Abschluss der Arbeit am XEE Projekt geschlossen werden.

Das Nachfolgende Diagramm zeigt die Verteilung der Operationen auf der TextArray Implementation einzeln aufgeschlüsselt nach Längen der Einfügung – in diesem Bereich bestimmt sich die Häufigkeit stark nach typischen Längen der Worte, die verwendet wurden zur Generierung des xmlgen Dokumentes. Dies Worte entstammen beim xmlgen Projekt dabei einer Wortliste von natürlichsprachlichen englischen Texten.



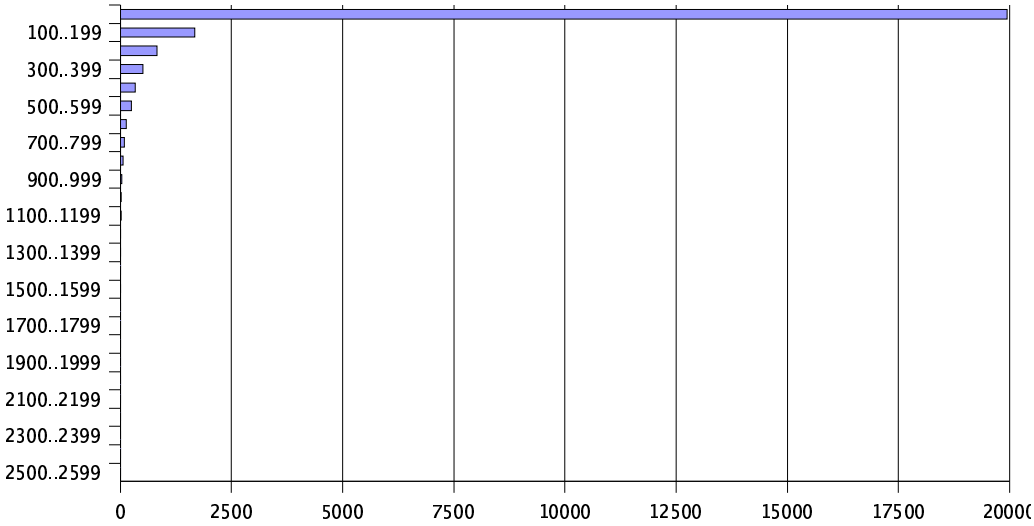
1-Bereiche Aufrufe mit 1..25

Hier die Verteilung im Bereich 0..250 der Längen je Einfügung. Durch die Zusammenfassung in Bereich zu je zehn ergibt sich eine deutlich logarithmische Kurve.



10-Bereiche Aufrufe mit 1..260

Die Betrachtung noch größerer Bereiche zeigt, dass es keine Auszeichnung mit Längen größer als 1700 gibt. Die Längen von Einfügungen ist konzentriert im Bereich der Blockgröße des Sekundärspeichers.



Wort-länge

Anzahl Aufrufe

100-Bereiche Aufrufe mit 1..2600

Insgesamt liegen für das 0.02 Dokument 23911 Einfüge-Operationen vor. Durch Betrachtung der Summen der Einfüge-Operationen ergibt sich folgende Prozenttabelle als Wahrscheinlichkeiten der Länge von Einfüge-Operation.

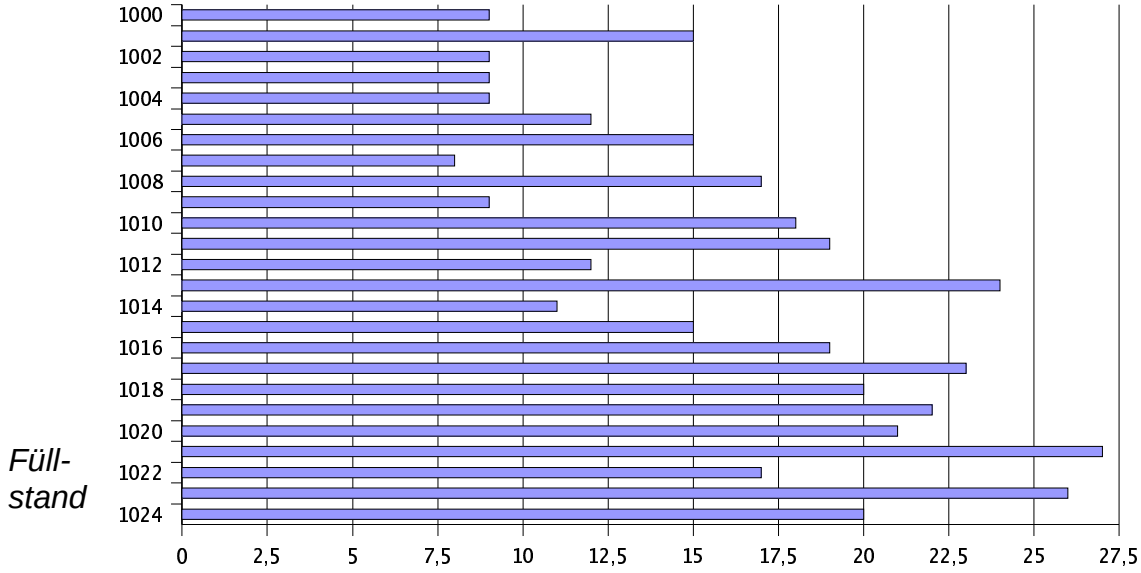
Länge der Einfügung	Summe	Prozent
0..125	20560	86,99%
0..250	22096	92,41%
0..500	23285	97,38%
0..1000	23844	99,72%
0..2000	23911	100,00%

Hier wird ersichtlich, dass die TextArray Implementation bestrebt sein muss, gerade die Einfügungen möglichst gut zu unterstützen, die sich häufig deutlich unterhalb der Blockgröße des Sekundärspeichers bewegen. Die vorliegende Variante füllt einen Block immer mindestens zur Hälfte, denn wenn die Einfügung nicht in den eine Block des Startpunktes passt, so wird der nächste Block betrachtet und wenn die Daten sich nicht auf diese beiden verteilen lassen, so wird ein neuer Block eingebunden und die Daten jeweils gleichmäßig verteilt.

Wenn man nur die Daten des Blockes am Einfüge-Startpunkt betrachtet und sofort neue Blöcke einbindet, so ergibt sich nach dem Einlesen der Datei ein gleichmäßiger Füllgrad mit einem Maximum bei 2/3 der Blockgröße – die Abweichung von diesem Mittel bewegen sich nach der Verteilung der Längen der Einfüge-Operationen modulo halber Blockgröße. Diese Beobachtung wurde bei der ersten Variante der TextArray Implementation gemacht. Die jetzt verwendete Variante dieses Algorithmus betrachtet jedoch davor immer zuerst

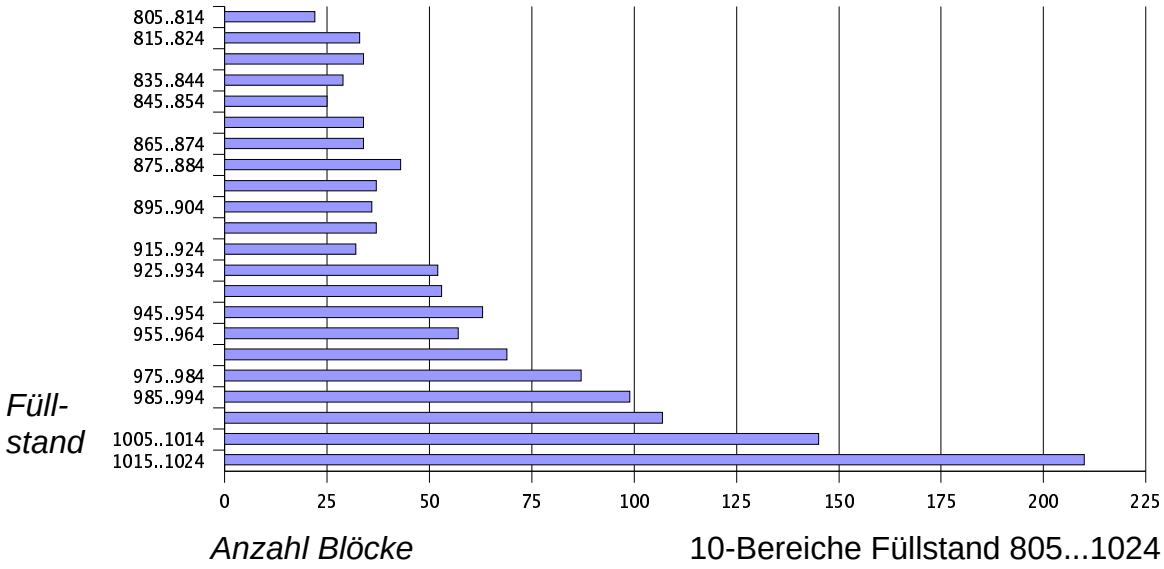
noch den nachfolgenden Block und verteilt die Summe der Daten aus beiden Blöcken und der Einfüge-Operation gleichmäßig. Dadurch ergibt sich nach dem ersten Einlesen der xmlgen Datei ein Füllgrad, der stärker bestrebt ist, die Blöcke ganz zu füllen, und erreicht gehäuft auch das Maximum eines voll genutzten Blocks.

Die nachfolgenden Tabellen beziehen sich wiederum auf das Einlesen einer Datei, die mit dem xmlgen Tool erstellt wurde und eine Größe von 2 Megabyte hat. Gemessen wird die Anzahl der Blöcke mit einem bestimmten Füllgrad, hier die Anzahl der Bytes in einem Block der Größe 1024. In der ersten Tabelle zeigt sich, dass die Werte stark ausschlagen, ein visueller Vergleich mit dem Diagramm zu den typischen Längen der Einfügungen zeigt jedoch, sich diese sich hier niederschlagen. Gerade am oberen Rand der Füllstände ist dies ersichtlich, da sich mehrere Einfügungen nicht ausgleichen sondern zu einem Split führen, dem Aufteilen des Textabschnitts eines Blocks auf zwei Datenblöcke.

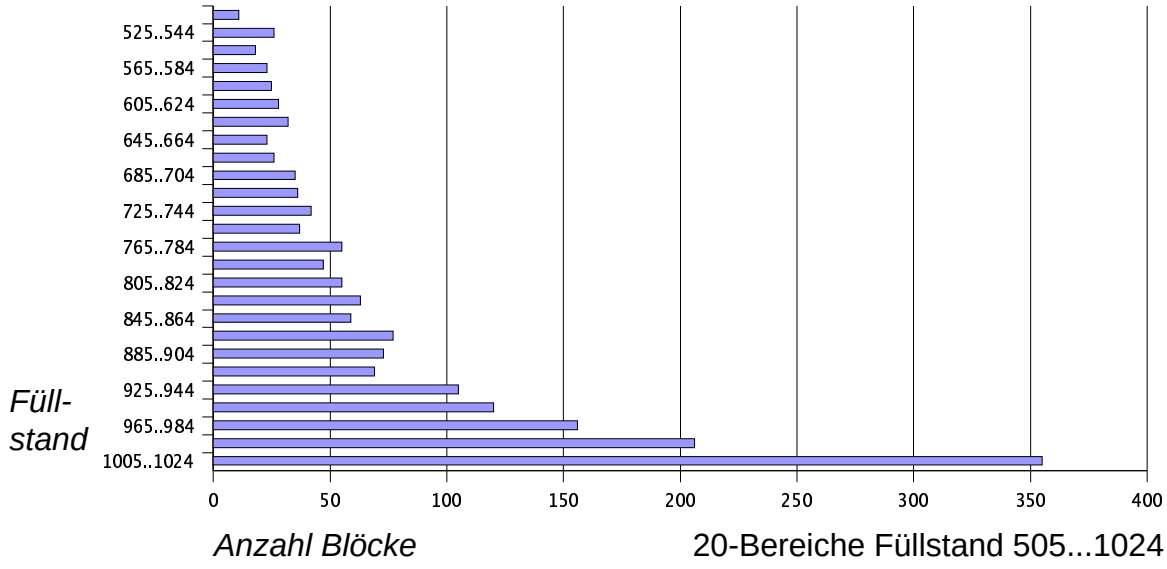


1-Bereiche Füllstand 1000...1024

Fast man die Bereiche zu Gruppen von Füllständen zusammen (hier je 10), so zeigt sich eine starke Häufung am oberen Ende. Die vorliegende Implementation ist daraufhin speziell angepasst, dass das erste Einlesen eines TextArray via XEE dazu führt, dass die Blöcke des Sekundärspeicher möglichst voll gefüllt sind. Es kann erwartet werden, dass spätere Veränderungs-Operationen im Textinhalt die Kurve verschieben, und den für B*-Bäume typischen Wert von 2/3 annehmen, bei einem 1024er Block also etwa 683.



Die letzte Tabelle zeigt die Werte zusammengefasst auf den gesamten Bereich, vom Maximum des Blockgröße bei 1024 bis hinunter zum Minimum bei 50% Füllstand der Blockgröße. Die Bereiche sind hier zu je 20 Füllständen zusammengefasst, die oberen fünf Balken stehen dabei grob für Füllstände besser als 90%. Auf der Wertachse ist die Anzahl eingetragen, und so sieht man leicht, dass mehr als die Hälfte der TextArray Blöcke besser als 90% gefüllt sind, direkt nach dem ersten Einlesen eines XML Dokumentes, das von xmlgen generiert wurde.



(35) Veränderungs-Operation auf Feldern

In dieser Phase noch keine Erkenntnisse. Dies kann erfolgen, wenn eine xupdate Funktion oder ähnliches zur Verfügung steht. Messungen dazu müssen an anderer Stelle erfolgen. Hier ist insbesondere die Zeitmessung zu beachten, die zum Auffinden und Weitersetzen von Positionen im TextArray sich ergeben.

Bei Änderungen von Längen der TextArray Abschnitte wird darauf zu achten sein, wie dieses in den Knoten der AST Hierarchie nachvollzogen wird. Es verschiebt sich dabei die logischen Position der Textelemente. Dies führt zu Veränderungen von Werten sowohl im Zuordnungsbaum der TextArray Implementation wie auch der Werte in der AST Implementation, die die TextArray Positionen der Auszeichnungsbereiche bereitstellen.

(36) Ergebnis

Die Implementation ist der Aufgabenstellung angepasst, soweit dies die derzeitig möglichen Messungen aufzeigen. Einige Details der Implementation können geändert werden, soweit in der weiteren Entwicklung des XEE Projektes vermutet werden kann, dass dieses Vorteile bringt.

Weitere Merkmale, die hinzugefügt werden können, ist etwa die Auftrennung der Datenhaltung des Zugriffsbaumes von den Datenblöcken um beispielsweise unterschiedliche Blockgrößen anwenden zu können, oder ein jeweils angepasstes Cache System aufzusetzen, dass die Blöcke im Speicher bereitstellt. Weiter dazu im nächsten Abschnitt.

(37) AST/TA Vorteile

Der AST/TA Ansatz trennt die gemischte Struktur des XML Dokuments auf und ermöglicht die schon erforschten und optimierten Algorithmen zu übernehmen, die für den Bereich des Information Retrieval und der Datenbanksysteme entwickelt wurden. Durch die Trennung bilden Operationen des Abfragens und Veränderns in den meisten Fällen nur auf einen der Teile ab: den AST mit der Strukturinhalten oder dem TA mit den Textinhalten.

Die Anfragen des Textinhaltes können Algorithmen aus dem Information Retrieval übernehmen, insbesondere Such-Operationen des Text Mining [SimpTR]. Zu einer gefunden Textstelle lässt sich über die logische Position im TextArray der Zugriffspfad bestimmen. Der Zugriffspfad mit enthaltenen Auszeichnungen und Attributen beschleunigt die Kategorisierung der Textstelle und deren Bewertung hinsichtlich einer Suchanfrage des Information Retrieval.

Die verbreitetste Form des Text Mining auf einem TextArray sind reguläre Ausdrücke (regex) nach PCRE, Perl Compatible Regular Expressions [PCRE]. Bei Verwendung auf Sekundärspeicher muss nur der Datenzugriff angepasst werden. Dies ist weitaus weniger aufwändig und fehleranfällig als speziell angepasst regex-Maschinen, die XML mitparsen. Ansätze findet man etwa in der Bioinformatik, erweiterte Referenzen finden sich unter [MARTEL], ein Vergleich von Messungen unter [BIO'parse].

Die im Verlauf einer Aufbereitung von Texten gefundenen Stellen können markiert werden, indem in den AST Knoten eingefügt werden, wobei der Zustand des Text Parsers nicht verändert wird. Die Struktur-Veränderungen sind unabhängig von Suchen und Bewertungs-Operationen im Text.

Andererseits stellt der AST auch die Beschreibung der Zugriffsstrukturen einer hierarchischen Datenbank dar, deren Feldinhalte im TextArray gehalten werden. Ein

Verändern der Feldinhalte ist über eine effiziente Veränderungs-Operationen auf einem TextArray zu erreichen, die Struktur des AST wird dazu nicht verändert. Effiziente Implementationen von Textspeichern in Hauptspeicher und auf Sekundärspeicher sind gut beschrieben. Im folgenden Abschnitt dieser Arbeit wird an die Grundlagen einer Implementation herangeführt, wie sie für das XEE Projekt errichtet wurde.

Die Wiederverwendung von gut erforschten Algorithmen aus dem Text Mining (etwa Suche und Indexierung) und hierarchischer Datenbanken sowie der neueren Ansätze zur direkten Arbeit mit XML ermöglichen sehr effiziente Implementationen für Teile oder Kombinationen der Grundoperationen im AST/TA: der Struktur Anfrage, dem Struktur Veränderung (je im AST), der Text Suche und Text Veränderung (je im TA).

4. Ausblick

(38) Angepasste Operationen

Ein Ziel der Implementation eines speziellen TextArray im XEE Projekt ist die Befähigung zu speziell angepassten Operationen auf dem Textinhalt. Derartige Operationen sind derzeit nicht definiert. Sie sollen der schnellen Suche im Textinhalt dienen.

Zu den klassischen Operationen auf dem Textinhalt gehören die Perl-Compatible Regular Expressions [PCRE] und andere Operationen, die in dieser Arbeit angesprochen wurden. Am Beispiel der PCRE zeigt sich, dass die Anpassung einer schon vorhandenen Implementation vergleichsweise einfach ist, da keine XML Auszeichnungen im Text mehr beachtet werden müssen.

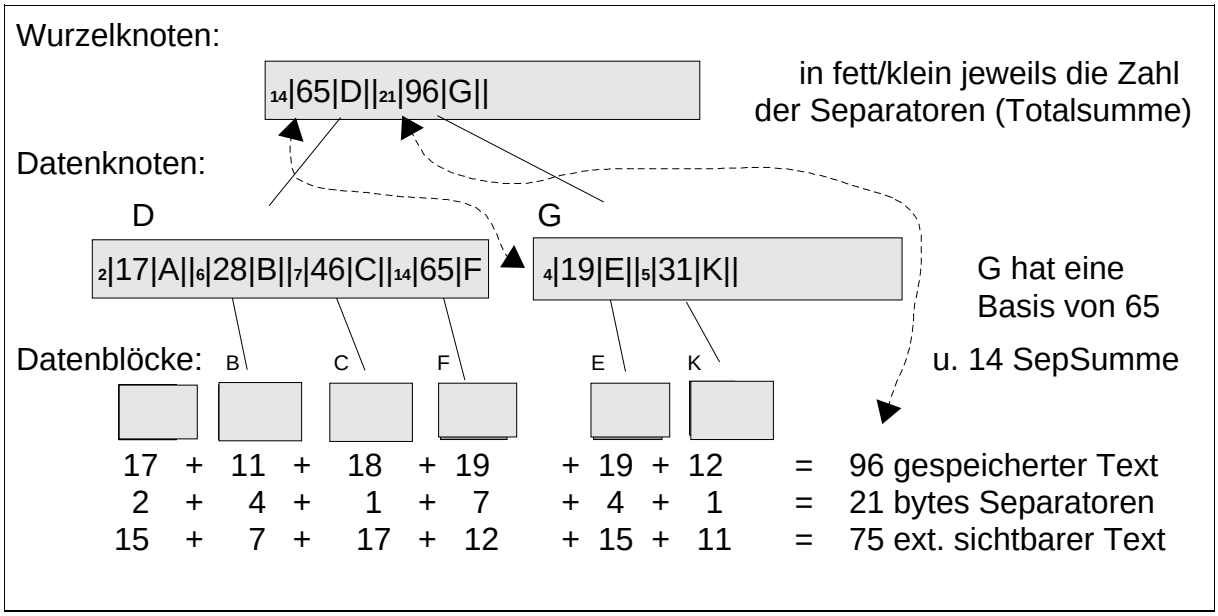
Stattdessen kann der byte-weise Zugriff der PCRE Maschine umgesetzt werden auf den Zugriff auf Zeichenpositionen im TextArray. Das Vorwärts- und Zurücksetzen im Text funktioniert auf dem TextArray des XEE Projektes genauso wie bei einem reinen Text. Der Zeichenabstand wird nicht durch XML-Auszeichnungen unterbrochen und Textteile in verschiedenen angrenzenden Auszeichnungen liegen logisch beieinander. Das physische Auffinden der TA Implementation kann über die Speicherpositionen abstrahieren.

Der verbleibende Unterschied einer PCRE Implementation auf Texten im Hauptspeicher verbleibt die Existenz der Separatoren im TextArray, die den Byteabstand von Textelementen beeinflussen ohne in den logischen Textabstand des ursprünglichen Textes einzugehen. Dabei zeigt sich, dass PCRE Maschinen darauf vorbereitet sind, indem sie sogenannte zero-width assertions kennen, die im Suchmuster vorkommen, aber keine Textelemente treffen müssen. In den PCRE Suchmustern gibt es etwa die spezielle Angabe „\b“, die eine Position zwischen einem Wort und Leerraum bezeichnet. Diese kann modifiziert werden, um Separatoren zu treffen und so eine echte Bytelänge zu gewinnen, ganz analog zu Angaben wie „\s*“ die eine beliebig langen Leerraum treffen, aber auch null lang sein können.

(39) Vorhaltung Separatorsummen

Die gegenwärtige Form der Verbindung von AST und TA verwendet Byte-Positionen und Byte-Längen zur Bestimmung von Textabschnitten. Dies funktioniert für XEE Operationen korrekt und sollte auch bei UTF-8 Kodierung kein Problem darstellen. Die Separatoren jedoch sind eine spezielle Eigenheit der AST/TA Implementation im XEE – und müssen für Operationen herausgezogen werden, die damit nicht umgehen können, insbesondere Operationen, die über äußere Zugriffe an das XEE angeschlossen sind.

Eine der wichtigen Operationen ist dabei, dass zu einem gefunden Textabschnitt dessen Position und Länge in einem Wert ohne Separatoren gefunden werden kann. Es ist ungünstig, zur Bestimmung der Anzahl der Separatoren in einem Textabschnitt dessen Speichertext durchzugehen und die Separatoren zu zählen. Neben der Möglichkeit, die Anzahl der Separatoren im AST zu speichern, ist es sinnvoll, die Anzahl der Separatoren in den Zuordnungstabellen der TA Implementation zu speichern.



Dies erfolgt dort ganz parallel zu den Positionsschlüsseln – diese beginnen in der vorliegenden Variante und stellen den Textabstand zum Anfang des TextArray dar. Dieser Textabstand wird einschließlich der Separatoren gerechnet und jeder Textblock des Sekundärspeichers besitzt den Textabstand seines Anfangs im Zuordnungsbaum. Zusätzlich ordne man jedem Anfang eines Textblocks die Summe der Separatoren zu. Zieht man diese Zahl von der logischen Position ab, erhält man die Position für ein Zugriffs-Modell ohne Separatoren.

Hat man einen Textabstand in der Mitte eines physisch zugeordneten Blocks, so muss dieser zur Separatorbestimmung zwar geöffnet werden, um die Separatoren von Anfang des Blocks bis zur Zugriffsposition zu zählen, jedoch ergibt sich der Gesamtwert mit der gespeicherten Separatorsumme für den Anfang des Blocks. Mit Hinblick auf mehrere AST über demselben TA, wie später beschrieben, ist diese Erweiterung auch effizient, da sonst bei Änderungen des Textinhaltes, und damit möglicher Änderungen der Separatorenanzahl, mehrere AST durchgegangen werden müssen, auch wenn sich die Textpositionen nicht verändert haben.

(40) Mehrere AST über einem TA

Die Trennung von AST und TA ermöglicht, dass über dem gleichen Textinhalt mehrere Zugriffsstrukturen errichtet werden können. Dies ist im besonderen beim Information Retrieval vorteilhaft, bei der gefunden Muster zu Textbereich Kategorisierung vermerkt werden, sodass nachfolgende Abfragen nach diesen Mustern beschleunigt werden. Diese Kategorisierungen und Bewertung sind abhängig von dem Ziel der späteren Abfragen und dabei selbst hierarchisch.

Verschiedene Kategorisierungen jedoch können sich überlappen und sind damit nicht mehr im XML Format selbst darstellbar – ein XML Auszug ist dann nur noch eine Sicht auf den Dateninhalt bezüglich des dafür angewandten Auszeichnungssystems. Das AST/TA Modell ermöglicht, diese Form in seiner Implementation selbst vorzuhalten – zu Fundstellen im Textinhalt (des TextArray) lassen sich dann schnell die Zugriffspfade in

mehreren Strukturangaben finden und deren Informationen in der Bewertung eines Information Retrieval Prozesses verwenden.

Als Beispiel fanden sich die Zugriffskategorisierungen der Bioinformatik, bei der chemische Verbindungen mehrere biologische Funktionen erfüllen können, nicht nur als Gesamtmolekül sondern mit Abschnitten des Proteins. Diese Funktionsabschnitte können sich grundsätzlich überlappen. Bei Fragestellungen an entsprechende Datenbanken sind solche Überlappungen aber möglicherweise gerade das Ziel der IR Anfrage. Der AST/TA Ansatz kann dieses effizient unterstützen.

Dabei müssen Veränderungen am TextArray mit mehreren aufgesetzten AST Bäumen synchronisiert werden, soweit Änderungen etwa an Positionen oder Separatoren aufgetreten sind. Weiterhin ist zu überlegen, ob ein AST Abschnitte mehrerer Textinhalte verwalten kann, etwa in der Kombination unter dem gleichen Strukturmodell aus Teilquellen. Die AST und TA Teile können jeweils Operationen auf hierarchischen Strukturen oder linearen Textfeldern wiederverwenden, die Verwaltung bei Veränderungs-Operationen muss jedoch genauer erforscht werden.

(41) Synchronisation / Mehrfachzugriff

Neben der Verwaltung mit mehreren AST Hierarchien über einem TextArray ist auch der Mehrfachzugriff auf eine AST/TA Implementation abzuklären. Es können dabei Mechanismen der hierarchischen Datenbanken zum Locking von Teilen verwendet werden. Durch die Trennung AST und TA erscheint es möglich, dass je nach Lock bestimmte andere Operationen nicht ausgeschlossen werden müssen.

Als Beispiel findet sich wieder eine Proteindatenbank der Bioinformatik. Bei einer Neueinfügung einer Proteinsequenz kann diese unmittelbar zur Verfügung stehen, die Kategorisierung jedoch schrittenweise nachfolgen. Dabei können Prozesse des Information Retrieval vielfach parallel erfolgen. Einfügungen der Ergebnisse als Unterstrukturierung im AST beeinflussen nicht den Lock auf den Textabschnitt.

Diese Parallelität gilt umso mehr, wenn mehrere AST über dem gleichen Textabschnitt errichtet werden: alle Operationen können solange parallel laufen, wie am Inhalt des Textabschnitts nichts geändert wird. Spätere Änderungen können dann in der Struktur und deren TextArray Referenzen nachgeführt werden und auch hierbei parallel erfolgen.

(42) Verwaltung von Zugriffsrechten

Die Verwaltung in getrennten AST und TA Teilen ermöglicht, dass besonders große Datenbestände vorgehalten werden können. Auch können Textinhalte und Strukturen aus verschiedenen Teilen zusammengeführt werden, denn es ist möglich, Text Mining nur auf Teilabschnitten oder über alle Bereiche auszuführen. Gerade die Wiederverwendung von Strukturen in Teilen eines Zugriffsbaumes ist möglich, und kann bei der Suche nach relevanten Informationen berücksichtigt werden.

Die Zusammenführung von Datenbeständen macht es erforderlich, Zugriffsrechte auf Teile der Daten zu verwalten und zu prüfen. Anteile der Daten mögen so zwar einen passenden Inhalt und Struktur zu einer Anfrage haben, sollen jedoch nicht abrufbar sein, abhängig von den Zugriffsrechten des Nutzers und den Datenteils, der eine Textstelle enthielt.

Diese Zugriffsrechte können auch unterschieden werden nach den Operationen, die ermöglicht oder abgelehnt werden sollen – so etwa die weit verbreiteten Operationen nach Ansicht, Modifikation des Inhalts, und Löschen einschließlich der Struktur, in Analogie zu Lesen, Schreiben und Löschen von Dateien in einem Dateisystem. Auch in verbreiteten

Dateisystemen treten dabei Fragen auf, wenn etwa ein Text mehrfach referenziert wird mit unterschiedlichen Zugriffsbits (dort: hardlinks, inode bits, bits on parenting directory).

5. Literaturverzeichnis

[ASTTA] Access Support Tree & TextArray: Data Structures for XML Document Storage - Dieter Scheffner - 12/2001
http://www.dbis.informatik.hu-berlin.de/pub/papers/techreports/HUB-IB-157.pdf

[BIO'parse] ParserComparison @ BioPython Wiki - Andrew Dalke
http://www.biopython.org/wiki/html/BioPython/ParserComparison.html

[CSS2] Cascading Style Sheets, level 2 - W3C
http://www.w3.org/TR/REC-CSS2

[DB2XML] DB2 XML Extender - IBM .com
http://www.ibm.com/software/data/db2/extenders/xmlext/

[DB2XML-] DB2XML - Making legacy data accible for XML applications - Volker Turau
http://www.informatik.fh-wiesbaden.de/~turau/ps/legacy.pdf

[DB2XML+] DB2XML tool - Transforming relational databases into XML documents - Volker Turau
http://www.informatik.fh-wiesbaden.de/~turau/DB2XML/

[DT4DTD] Datatypes for DTDs (DT4DTD) - W3C - 1.0 in 01/2000
http://www.w3.org/TR/dt4dtd

[exlnDB] XIS - eXtensible Information Server - eXcelon Corp
http://www.exln.com/products/xis/

[EXODUS] Object and File Management in the EXODUS Extensible Database System - Michael J. Carey, David J. DeWitt, Joel E. Richardson, Eugene J. Shekita - VLDB 1986
http://www.vldb.org/conf/1986/P091.PDF

[Fuhr'IR] Information Retrievial - Norbert Fuhr - VL-Script, Uni Trier
http://ls6-www.informatik.uni-dortmund.de/ir/teaching/courses/ir/script/irskall.ps.gz

[Intel'P3] Intep Pentium III Processor - Intel .com
http://www.intel.com/pentiumiii

[Intel'P4] Intel Pentium 4 Processor - Intel .com
http://www.intel.com/pentium4

[KL'XQ] XML-Anfragesprache - Karsten Lücke - 01/2001
http://www.informatik.hu-berlin.de/~luecke/kl_xmlquery.cover.ps.gz

[MARTEL] Martel: Bioinformatics file parsing made easy - Andrew Dalke
http://biopython.org/~dalke/Martel/ipc9/

[ORA9XML] Oracle XML DB - Oracle .com
http://otn.oracle.com/tech/xml/xmldb/

[PCRE] Perl Compatible Regular Expressions - Philip Hazel
http://www.pcre.org

[PN] Proximal Nodes: A Model to Query Document Databases by Contents and Structure - Gonzalo Navarro, Ricardo Baeza-Yates - 1997
ACM Transactions on Information Systems 15(4)

[SAX] SAX - Simple API for XML - http://www.saxproject.org/

[SGMLH] The SGML History Niche - Charles F. Goldfarb http://www.sgmlsource.com/history

[SimpTR] Simple, proven approaches to text retrieval - SE.Robertson, K.Sparck Jones - 05/1997 http://www.ftp.cl.cam.ac.uk/ftp/papers/reports/TR356-ksj-approaches-to-text-retrieval.html

[Tamino] Tamino XML Server - Software AG http://www.softwareag.com/tamino/

[VR'IR'79] Information Retrievial - C.J.vanRijsbergen - 1979 http://www.dcs.gla.ac.uk/Keith/Preface.html

[W3XML] Extensible Markup Language (XML) - W3C http://www.w3.org/XML

[WWW] World Wide Web Consortium - W3C http://www.w3.org

[XEE] The XML Query Execution Engine (XEE) - Dieter Scheffner, J.C.Freytag - 03/2002 http://www.dbis.informatik.hu-berlin.de/pub/papers/techreports/HUB-IB-158.pdf

[X-Hive] X-Hive/DB - XML database - X-Hive Corp http://www.x-hive.com/

[XMLDBs] *Teamgeist! - Entwicklung neuer Standards für XML-Datenbanken* - Lars Martin - Linux-Magazin 04/2001 http://www.linux-magazin.de/Artikel/ausgabe/2001/04/XMLDB/xmldb.html

[XMLnDB] XML and Databases - Ronald Bourret - 1999-2002 http://www.rpbourret.com/xml/XMLAndDatabases.htm

[XML'RDB] XML Representation of a Relational Database - W3C - 07/1997 http://www.w3.org/XML/RDB.html

[XMLtypes] XML Schema Part 2: Datatypes - W3C - 05/2001 http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

[XPath] XML Path Language (XPath) - W3C http://www.w3.org/TR/xpath

[XQuery] XML Query - W3C http://www.w3.org/XML/Query/

[XUPDATE] XUpdate : XML Update Language - XML:DB Initiative http://www.xmldb.org/xupdate/