Universität Konstanz
FB Informatik und Informationswissenschaft
Bachelor-Studiengang Information Engineering

Bachelorarbeit

Iterative XQuery Anfrageverarbeitung in BaseX
Iterative “Query” Processing in BaseX

zur Erlangung des akademischen Grades eines
Bachelor of Science (B.Sc.)

Studienfach: Information Engineering

Schwerpunkt: *Datenbank-Technologie*

Themengebiet: Informatik der Systeme

von

Dennis Stratmann
(622495)

Erstgutachter:	Prof. Dr. Marc H. Scholl
Zweitgutachter:	Prof. Dr. Marcel Waldvogel
Betreuer:	Christian Grün
Einreichung:	14.09.2010

Zusammenfassung

Im Zuge der Verbreitung des Internets wachsen auch die Verbreitung des XML-Datenformats sowie die Größe der XML-Dokumente. Daher sind native XML-Datenbanken auf den Vormarsch, um die gespeicherten Informationen effizient abfragen zu können.

Diese Arbeit widmet sich dem Konzept der iterativen Anfrageverarbeitung von XQuery-Anfragen in einem XQuery-Prozessor. Dieses wird sowohl theoretisch beleuchtet sowie die praktische Umsetzung in dem Prozessor der BaseX XML-Datenbank aufgezeigt. Vorteile sind der theoretisch konstante Speicherverbrauch und eine teilweise bessere Performance gegenüber der herkömmlichen DOM-basierten Herangehensweise. Jedoch kann es nicht durchgehend gebraucht werden, da sogenannte *Blocking Operators* dieses Konzept in einer Query unmöglich machen.

Abstract

With the growing use of the XML data format as well as the increasing size of the XML documents itself, native XML databases become more and more interesting to query the stored information more efficiently.

This thesis is dedicated to the iterative Query processing concept for XQuery requests. This concept will be highlighted both theoretically and the practical implementation in the BaseX XQuery processor. It has the advantages of theoretical constant memory consumption as well as partial better performance compared to the traditional DOM-based approach. But it can't be used in all situations, because of the so called b*locking operators*, which makes this concept impossible.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation und Zielsetzung dieser Arbeit

Da das XML-Format ein Standard im Web geworden ist, sind effiziente und effektive Abfragen ein wichtiges Forschungsfeld. Die Existenz von mehr als fünfzig verschiedenen XQuery-Prozessoren zeigt das klare Interesse von Forschung und Industrie sich dieser Herausforderung zu stellen.

Da die meisten Daten anfangs noch relativ klein waren schien es eine einfache Lösung zu sein, die kompletten XML-Daten in den Speicher zu laden um sie dann erst abzufragen. Ein Beispiel ist DOM [1], das für solch eine Herangehensweise steht.

Als Abfragesprache diente erst XPath und später XQuery [2]. XQuery, das zurzeit in der Version 1.0 vorliegt, ist eine Erweiterung von XPath 2.0. Es hat in XML-Datenbanken die gleiche Funktion wie SQL in Relationalen Datenbanken. Doch durch die rasant wachsende Menge und auch Größe der XML-Daten, sowie der parallele Zugriff darauf, ist die traditionelle DOM-basierte Herangehensweise nicht mehr tragbar.

Diese Arbeit richtet sich an den Kern einer XML-Datenbank, den XQuery-Prozessor. Es wird gezeigt, wie die iterative Abarbeitung von XQuery-Anfragen umgesetzt wird und wann dieser Ansatz überhaupt genutzt werden kann, also welche Voraussetzungen von Seiten der XML-Daten erfüllt sein müssen, sowie die Eigenschaften der Query. Zudem wird auch die konkrete Umsetzung in BaseX [3] beschrieben und welche Besonderheiten diesen XQuery-Prozessor auszeichnen.

1.2 Iterative Anfrageverarbeitung

Das „Streaming“, „Pipelining“ oder einfach die iterative Anfrageverarbeitung ist ein bekanntes allgemeines Datenbank Konzept [4], das inzwischen auch aus nativen XML-Datenbanken nicht mehr wegzudenken ist.

Wie die Synonyme schon vermuten lassen, lässt sich die iterative Anfrageverarbeitung in XQuery beispielsweise gut mit dem Streamen von Videos im Internet vergleichen. Diese Videos können während sie noch geladen werden schon angeschaut werden. Ähnlich ist es beim Pipelining. Besteht eine XQuery-Anfrage aus mehreren Expression, wird jedes Item einer Expression sofort ausgewertet. Handelt es sich um ein Ergebnis, wird dieses zur nächsten Expression weitergeleitet. Das garantiert einen theoretisch konstanten Speicherverbrauch und verhindert eine Nicht-Ausführbarkeit gewisser Anfragen mit speicherintensiven Zwischenergebnissen.

Ein nicht iterativer Ansatz dagegen kann mit einem Video verglichen werden, dass zuerst komplett geladen werden muss, bevor es gestartet werden kann. Es werden also zuerst alle Items einer Expression im Speicher geladen, ausgewertet und erst dann an die nächste Expression weitergeleitet. Das kann zu großen Zwischenergebnissen im Speicher führen, die nicht unbedingt benötigt werden. Gerade wenn zum Beispiel eine *Descendant-or-Self*-Achse genutzt wird, könnte das schnell dazu führen, dass das Dokument als Ganzes, vielleicht sogar mehrmals, im Speicher liegt. Das ist bei kleinen Dokumenten selbst noch kein Problem. Beziehen sich jedoch mehrere Anfragen auf ein Dokument oder ist es zu groß, kann es zu Speicherproblemen kommen.

1.3 Problemstellung

Der Vergleich mit dem Videostreaming im vorherigen Kapitel hinkt aber insofern, dass zwar einige Expression einer Query iterativ bearbeitet werden können, andere jedoch nicht. Dadurch wird eine Query insgesamt nur teilweise iterativ bearbeitet.

Welche *Expression* dieses Kriterium nun erfüllen und aus welchen Grund wird im *Kapitel 3 „Pipelining“* erläutert. Hierbei wird besonders auf *Path Expression* und *Sequence Expression* eingegangen.

Im *Kapitel 5 „Relevante Forschungsarbeiten“* wird noch ein anderes Forschungsfeld, dass der transistenten XML-Daten, aufgezeigt, welches durch die andersartige Voraussetzung auch unterschiedliche Algorithmen benötigt.

1.4 BaseX als native XML Datenbank

BaseX [3] ist eine native, Open-Source XML-Datenbank, die an der Universität Konstanz am Lehrstuhl Datenbanken und Informationssysteme entwickelt wird. Es hat einen effizienten XPath/XQuery-Prozessor, unterstützt XPath/XQuery Full-Text, basiert auf einer Client/Server-Architektur und bietet ein ansprechendes interaktives Front-End.

BaseX dient dieser Arbeit als praktische Grundlage, in dem viele der hier theoretisch besprochenen Konzepte umgesetzt wurden, sowie als Testkandidat für Performance und Speicherverbrauch. Die Tests wurden teilweise mit XMark [5] durchgeführt und teilweise mit speziell für das Pipelining optimierten Queries.

Auf BaseX und dessen Funktionsweise, sowie vor allem die Implementierung des iterativen Ansatzes, wird im *Kapitel 2 „XQuery Prozessor in BaseX“* noch ausführlicher eingegangen. Die erzielten Ergebnisse und deren Interpretation werden im *Kapitel 4 „Benchmarkergebnisse“* vorgestellt.

2 XQuery Prozessor in BaseX

Um die im *Kapitel 3 „Pipelining“* vorgestellten Konzepte der iterativen Bearbeitung besser mit BaseX betrachten zu können, muss erst auf einige Besonderheiten von BaseX und dessen Architektur näher eingegangen werden. Hierzu dient dieses Kapitel.

2.1 Speicherung

Das p***re/post***-Encoding hat sich mit der Zeit als eine performante Lösung zur Speicherung der XML-Knoten herausgestellt [7]. Eine Variante dieses Encodings ist das **pre/size/level**-Encoding, bei der jeder XML-Knoten mit diesen drei Werten gespeichert wird. Diese Attribute repräsentieren eine Knotenidentifikation (*pre*), Anzahl an nachfolgenden Knoten (*size*) und die Tiefe des Knotens im Dokumentenbaum (*level*). Beide Varianten sind äquivalent, da *post = pre + size – level* ist, und kam in den früheren XML-Datenbanken zum Einsatz.

Auch BaseX nutzt eine Variante dieses Encodings – es speichert XML-Knoten allerdings in einer Kombination von ***pre/dist/size***-Triplets für jeden Knoten ab [6]. Hierbei steht *dist* für die Distanz zum *parent*-Knoten (siehe *Abbildung 1*).

Dieser Distanzwert ist relativ und nicht mehr absolut wie es noch bei der vorher genutzten Methode war. Daraus resultiert das es Update-invariant ist. Ein Subtree behält seine originalen Distanzwerte, wenn er verschoben oder sogar in ein anderes Dokument kopiert wird. Effiziente Updates sind auch mit ein wenig Aufwand in dem *pre/size/level*-Encoding möglich [7,8].

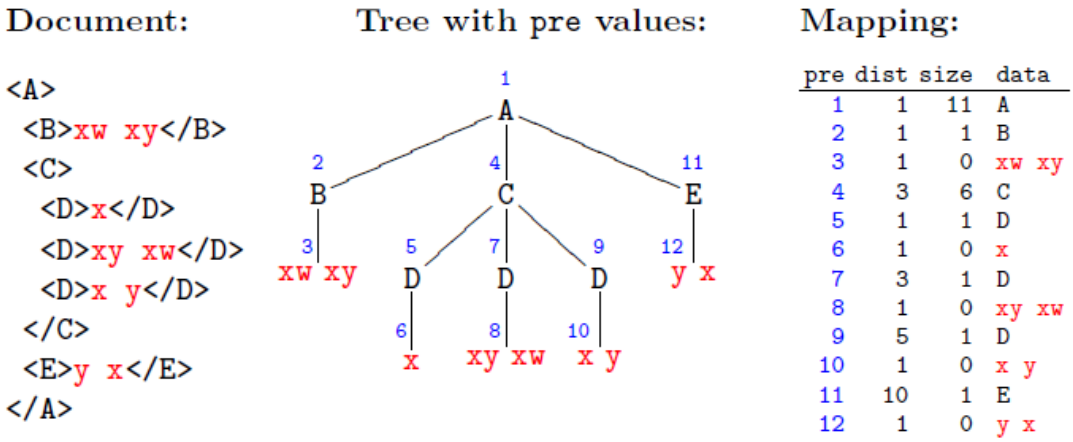


Abbildung 1: Dokument Encoding in BaseX (Quelle [6])

2.2 Ordnung, Sortiertheit und Duplikatfreiheit

XML ist per Definition ein geordnetes Datenformat. Die relative Ordnung von Elementen in einem XML-Dokument ist eine wichtige Eigenschaft und nennt sich *Document Order*. Es ist die Reihenfolge, in der Knoten bei der XML-Serialisierung eines Dokuments vorkommen [2]. Durch das pre/post-Encoding wird sichergestellt, dass XML-Knoten in *Document Order* eingelesen werden.

Dies wird gut in *Abbildung 1* verdeutlicht. Die Pre-Werte unter *Mapping* entsprechen der *Document Order*. Höhere Pre-Werte bedeuten „späteres" Vorkommen in der *Document Order*. Diese entsprechen dem Pfad, der im Baum entlang dieses Verfahrens gegangen wurde. Konkret heißt das, *A* ist in der *Document Order* vor *B* und dieser wiederum vor *C*. Im Kapitel *3.3.2.1 „Triviale Location Paths"* wird ein Beispiel vorgestellt, bei der durch einen simplen *Descendant-or-Self* mit gefolgtem *Child*-Schritt eine sich nicht in *Document Order* befindende Sequenz von Knoten entstehen kann.

Die Eigenschaft der *Document Order* macht XML zu einem guten Format, um Information in einer semantisch sinnvollen Reihenfolge darzustellen. So kann zum Beispiel die Abfolge von Ereignissen in einer Logdatei oder einem chirurgischen Protokoll geordnet aufgeführt werden.

Die formale Semantik von XQuery [12], und somit auch von XPath, verlangt, dass das Resultat einer Path Expression in *Document Order* sortiert und duplikatfrei ist. Daraus ergibt sich für XQuery-Implementationen, die sich strikt an diese formale Semantik halten, nach jedem *Location Step* zu sortieren und Duplikate zu eliminieren. Dies wiederum kann einen enormen Overhead erzeugen gerade wenn große XML-Dokumente abgefragt werden, den man jedoch teilweise durch geschickte Planung umgehen kann.

Dazu wird in *Kapitel 3.3.2.2 „Order Automat"* ein Ansatz in Form eines deterministischen Automaten vorgestellt der überflüssige Sortierschritte aus einer Path Expression entfernt. Der Automat ist gleichzeitig mit dem in *Kapitel 3.3.2.3 „No Duplicates Automata"* auch der Kernpunkt der iterativen Verarbeitung von Path Expression, da alle Pfade, die keine nachträgliche Sortierung und Duplikatentfernung benötigen direkt, für das Pipelining genutzt werden können.

XQuery arbeitet nun zum einen auf der Datenstruktur geordneter Bäume von XML-Knoten wie sie durch das pre/post-Encoding erstellt werden; zum anderen auf endlichen geordneten Folgen von Items (atomische Werte oder Knoten). Hier kommt dann die *Sequence Order* zum Tragen, die die Ordnung einer XQuery Expression in einer Item-Sequenz sicherstellt – ganz wie man es aus dem traditionellen SQL kennt. Zum Beispiel kann eine XQuery Expression eine Sequenz von Personen nach der Höhe des Gehalts sortieren.

Während Ergebnisse einer *Path Expression* immer in *Document Order* zurückgegeben werden müssen, ist die *Sequence Order* eine vom User gegebene Ordnung. Da Sequenzen also keine inhärente Ordnung haben ist folgendes Beispiel für die in *Kapitel 3.3.1 „Primary Expression – Function Calls“* vorgestellten kein Pipelineblocker:

(2 , 4 , 9 , 3 , 5 , 1)

Angenommen der Durchschnitt der Werte wird gesucht, dann ist die *Nicht-Sortiertheit* der Sequenz kein Problem. Auch eine Sequenz mit *Step*-Knoten benötigt keine Ordnung. Zum Beispiel führt in der folgenden Query

(//a, //c, //b)/self:node()

erst der letzte *Self Step* dazu, dass die Knoten nach *Document Order* sortiert werden.

2.3 Anfragebearbeitung

Bei der Abarbeitung einer Query durchläuft BaseX die vier in *Abbildung 2* gezeigten Phasen.

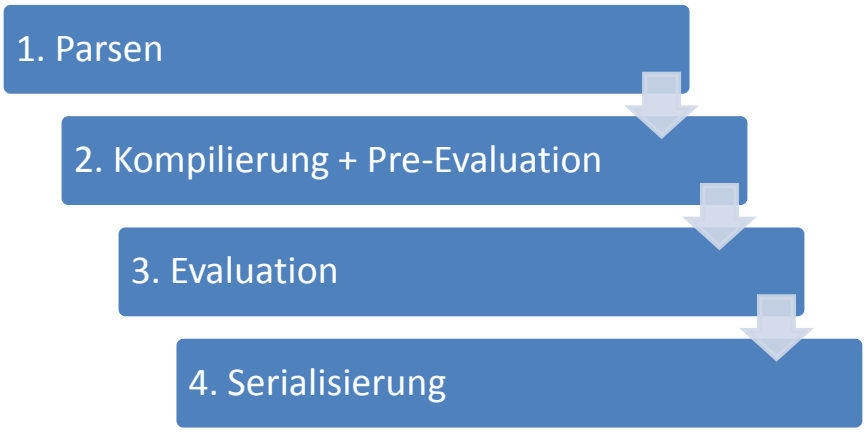


Abbildung 2: Schritte der Anfrageverarbeitung in BaseX

- *Phase 1: Parsen:*
  Die Query wird geparst und auf syntaktische Korrektheit überprüft. Bei falscher oder unvollständiger Eingabe der Query wird eine entsprechende Fehlermeldung ausgegeben. Bei entsprechender Aktivierung, gibt es die Option, den nicht optimierten Ausführungsplan auszugeben. Dieser wird dann aus der hier geparsten Query erstellt.

- *Phase 2: Kompilierung + Pre-Evaluation:*
  Im Kompilierungsschritt wird die Query bzw. einzelne Expression optimiert. Das kann zum Beispiel bedeuten, dass überflüssige Variablen entfernt, *Location Steps* umgeschrieben oder *FLWOR Expression* vereinfacht werden.
  Zudem werden einfache Expression oder Funktionen schon pre-evaluiert. Dabei kann es sich um einfache Rechenoperationen oder auch um Ausdrücke, die immer falsch bzw. wahr sind handeln.

- *Phase 3: Evaluation:*
  Es wird nun überprüft, ob die optimierte Query bzw. Expression iterativ abgearbeitet werden kann. Dies wird für jede Expression der Query für sich selbst entschieden. Ist dies möglich wird individuell eine iterative Evaluierungsmethode benutzt. Für alle anderen Fälle wird die normale Funktion herangezogen.

- *Phase 4: Serialisierung:*
  Mit der Ausgabe des Resultats wird ein Query Plan erstellt. Dieser zeigt an welche *Location Steps*, Expression und Funktionen benutzt wurden. Es gibt hier die Möglichkeit zum einen den Query Plan der optimierten als auch den schon in *Phase 1* beschriebenen ursprünglichen Queryplan auszugeben. Dieser Plan ist übersichtlich gestaltet und lässt einfach nachvollziehen, welche Schritte konkret unternommen wurden. Hieraus lassen sich auch Pipelining Schritte leicht ablesen.

2.4 Das Iterator Model

Generell gibt es in BaseX nur ein Iterator-Model. Es basiert auf der abstrakten Java-Klasse *Iter*. Zunächst wird die allgemeine Anwendung in einem Java-Code Beispiel gezeigt, bevor auf Einzelheiten sowie Vererbungen eingegangen wird.

```
1   private Iter methodeName (final QueryContext ctx)  throws QueryException
2   {
3       return new Iter()
4       {
5           final Iter eingabe_Iterator = expr.iter(ctx);
6
7           @Override
8           public Item next() throws QueryException
9           {
10              final Item item = eingabe_Iterator.next();
11              // führe Berechnungen durch
12              // wenn item == null keine weiteren Ergebnisse
13              // wenn item ein Ergebnis ist ...
14              return item;
15          }
16      }
17  }
```

Abbildung 3: Codebeispiel für iterative Verarbeitung

Wie in *Abbildung 3* gezeigt, wird in einer iterativ-verarbeitenden Methode ein neuer Iterator erstellt (Abb.3 - Zeile 3), der durch eine überschriebenen *next()* Methode (Abb.3 - Zeile 8) die Ergebnisse einer Query bzw. Funktion "just-in-time" auswertet (Abb.3 - Zeile 10-14).

```
1   private Iter methodeName (final QueryContext ctx)  throws QueryException
2   {
3       Iter sequenz = new Iter()
4       final Iter eingabe_Iterator = expr.iter(ctx);
5       Item item;
6
7       while ( (item = eingabe_Iterator.next()) != null)
8       {
9           // führe Berechnungen durch
10          // wenn item ein Ergebnis ist ...
11          sequenz.add (item);
12      }
13      return sequenz;
14  }
```

Abbildung 4: Codebeispiel für nicht-iterative Verarbeitung

Im Gegensatz dazu wird in dem materialisierenden Model in der Methode selbst das Ergebnis ausgewertet (Abb.4 – Zeile 7-11) und nicht in der *next()* Methode. Somit wird ein voll ausgewerteter Ergebnisiterator zurückgegeben (Abb.4 – Zeile 13), wie in *Abbildung 4* dargestellt.

In der *while*-Schleife (Abb. 4 – Zeile 7-12) werde alle Ergebnisse in dem Iterator *sequenz* gespeichert (Abb. 4 – Zeile 11). Da die Rückgabe *sequenz* selbst ein Iter-Objekt ist, hat es die Standard-Implementation der *next()* Methode (siehe *Abbildung 5*), die einfach immer das nächste Item zurückgibt ohne selbst irgendwelche Berechnungen durchzuführen. Im direkten Vergleich der beiden Herangehensweisen wird deutlich, dass bei der iterativ-verarbeitenden Methode die *Document Order* eine sehr wichtige Rolle spielt. Denn in *Abbildung 4* ist die fertige Sequenz vor der Rückgabe noch sortierbar. Dies ist jedoch nicht in *Abbildung 3* möglich. Es wird immer nur das nächste Ergebnis berechnet und es sind keine weiteren hilfreichen Informationen über die Ordnung oder Duplikatfreiheit der verbleibenden Items vorhanden.

Da ein Iterator verschiedene Inputs haben kann ist die Klasse *Iter* in BaseX mehrmals überladen, wie in *Abblidung 5* deutlich wird.

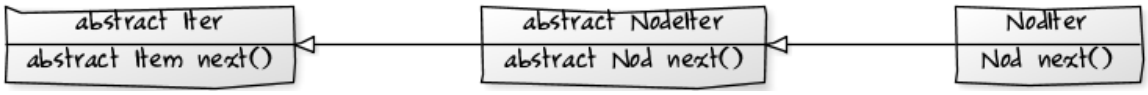


Abbildung 5: UML Klassendiagramm der Iterator Klassen

Der wesentliche Unterschied zwischen der *Iter*- und der *NodeIter*-Klasse ist also, dass Letztere für XML-Knoten gedacht ist. Die nochmals überladene *NodIter*-Klasse ist zudem für spezielle Fälle, in dem Duplikate ignoriert werden können.

Diese in *Kapitel 2* dargelegten Grundlagen des XQuery-Prozessors in BaseX werden nun im nächsten Kapitel weiter vertieft, auch im Hinblick auf allgemeine Pipelining-Konzepte. Doch vor allem werden die wichtigsten *XQuery Expression* beispielhaft erklärt sowie deren praktische Umsetzung erläutert.

3 Pipelining

Schon 1993 wurde das Konzept der iterativen Anfrageverarbeitung [4] beschrieben und setzte sich schnell in den Datenbanksystemen durch. Mit einer der Gründe dafür, zeigt die Gegenüberstellung der Vor- und Nachteile im folgenden Kapitel.

3.1 Vor- und Nachteile

Im Gegensatz zu nicht iterativen Ansätzen, bei denen Zwischenergebnisse komplett im Speicher gehalten werden, wird durch das einzelne Abarbeiten jedes Items eines Zwischenergebnisses und der direkten Weitergabe ein theoretisch konstanter Speicherbedarf garantiert. Dies verhindert auch gleichzeitig, dass ein materialisiertes Zwischenergebnis zu groß für den Speicher ist und dadurch die Anfrage nicht bearbeitet werden kann.

So ist bei simplen Datenbankoperationen zwar mit leichtem Overhead zu rechnen, doch wenn große Zwischenergebnisse und kleine Ergebnismengen zu erwarten sind, kann eine iterative Bearbeitung klare Performancevorteile bringen.

```
doc(“book.xml“)/article/chapter
```

Abbildung 6: Beispiel Query zur Verdeutlichung des Vorteils beim Pipelining

Bei der Bearbeitung der Query in *Abbildung 6*, wird für jedes gefundene *article*-Element eine Kindachse namens *chapter* gesucht. Derartige Elemente werden der Ergebnismenge hinzugefügt. Jedoch wird zu keiner Zeit ein Zwischenergebnis von allen *article*-Elementen gebildet.

Angenommen, es werden nur die ersten Items eines Zwischenergebnisses benötigt, legt die iterative Abarbeitung nahe, den Rest des Zwischenergebnisses erst gar nicht zu bearbeiten. Wie leicht zu erkennen ist, kann dies zu einem großen Performance-Vorteil führen, wenn ansonsten tausende Items hätten ausgewertet werden müssen.

Ein Nachteil dagegen ist die komplexere Entwicklung von Software mit Pipelining-Unterstützung. Diese rührt zum einem daher, dass die iterative Umsetzung nicht immer offensichtlich ist, was einer der Gründe für diese Bachelorarbeit ist; zum anderen liegt es auch daran, das gleichzeitig immer noch die materialisierende Methode umgesetzt werden muss, falls eine Methode wegen der Eingabedaten oder andere Gründen nicht iterativ arbeiten kann.

3.2 Voraussetzungen

Auch wenn sich das Paper mit dem Titel *„A Fully Pipelined XQuery Processor“* [13] vielversprechend anhört, ist diese Umsetzung in BaseX nicht möglich. Denn jener XQuery-Prozessor bezieht sich auf Datenströme, die in Echtzeit ausgewertet werden müssen und deren Daten nur einmal zur Verfügung stehen. Einige Anforderungen in BaseX lassen sich aber nicht umgehen und damit auch kein vollständiges Pipelining zu.

Wie in *Kapitel 2.2 „Ordnung, Sortiertheit und Duplikatfreiheit“* schon erläutert, wird eine *Pipeline* erst von sogenannten *Blocking Operators* gebrochen. Einer dieser Operatoren ist zum Beispiel das Sortieren. Muss ein Zwischenergebnis sortiert werden, ist dies ohne vorherige Auswertung aller Items logischerweise nicht möglich.

Ähnliches gilt für die Duplikatfreiheit. Einige Expression, wie zum Beispiel die *Union Expression*, können zwar mit Duplikaten iterativ bearbeitet werden, jedoch ist keine Garantie für die Duplikatfreiheit im Ergebnis. Und das entspräche nicht der *XQuery Semantik* [5].

Wie später noch im *Kapitel 3.3.2.2 „Order Automata“* und *Kapitel 3.3.2.3 „No Duplicates Automata“* gezeigt wird, müssen manche Queries nicht nach jedem Schritt sortiert werden, so dass einige Sortierschritte und Duplikatentfernungen überflüssig werden. Im sortierten Zustand kann dann das Pipeline-Verfahren fortgeführt werden.

3.3 XQuery Expression

Jede XQuery-Query enthält eine oder mehrere Query Expression. Laut [2] gibt es folgende Expression:

- *Primary Expression*
- *Path Expression*
- *Sequence Expression*
- *Arithmetic Expression*
- *Comparison Expression*
- *Logical Expression*

- *Constructors*
- *FLWOR Expression*
- *Ordered and Unordered Expression*
- *Conditional Expression*
- *Quantified Expression*

In dieser Arbeit wird das Hauptaugenmerk auf die *Primary Expression*, *Path Expression* und *Sequence Expression* gelegt. Zu Beginn werden die *Function Calls* aus der *Primary Expression*-Klasse hervorgehoben, um eine Grundlage zum weiteren Verständnis zu bilden.

3.3.1 Primary Expression - Function Calls

Die Liste der *built-in* XQuery-Funktionen besteht aus über 100 Funktionen [11]. Daher werden hier nur einige wenige betrachtet. Besonders interessant sind Funktionen aus den folgenden drei Bereichen:

Aggregate-Funktionen:

- fn:avg()
- fn:max()
- fn:min()
- fn:sum()

Accessor-Funktionen:

- fn:position()
- fn:last()

Sequence-Funktionen:

- fn:distinct-values()
- fn:index-of()
- fn:insert-before()
- fn:remove()

Dieser Auszug an Funktionen soll verdeutlichen, dass diese Funktionen zum einen teilweise unterschiedlichen Input erwarten, der nicht zwangsweise sortiert sein muss, zum anderen auch sehr einfach sein können, wie schon im *Kapitel 2.2 „Ordnung, Sortiertheit und Duplkatfreiheit“* kurz aufgezeigt wurde.

Gerade die *Aggregate*-Funktionen bestechen durch ihre simple Art und machen einen nicht iterativen Ansatz geradezu unlogisch. Als Beispiel soll *fn:max()* dienen. Es würde keinen Sinn machen, alle Items in den Speicher zu laden, um sie dann erst zu vergleichen. Stattdessen wird jedes Item geladen, mit dem bisherigen Maximum verglichen und sofort wieder verworfen. Auch die Voraussetzung der Sortiertheit muss nicht gegeben sein, da sich die Funktion nicht auf die *Document Order* bezieht sondern auf die Semantik der Sequenz. Doch könnte das Wissen über die Sortiertheit hier einen Vorteil in sich bergen, da so nicht alle Items verglichen werden müssen. Ähnlich simpel ist der Rest der *Aggregate*-Funktionen aufgebaut.

Bei den *Accessor*-Funktionen könnte eine ähnliche Herangehensweise genutzt werden. *fn:position()* und *fn:last()* suchen Positionen und lassen sich daher leicht iterativ implementieren, indem bis zur gesuchten Position iteriert wird. Jedoch hat BaseX eine andere Methode gewählt. Hier werden die Informationen der QueryContext-Klasse genutzt. Sie ist eine abstrakte Klasse, die die Architektur der kompilierten Eingabequery bereitstellt. Wird also das letzte Item (*fn:last()*) gesucht, kann man über die aktuelle Größe des Querykontextes dieses Item abfragen. Wird das Item an einer bestimmten Position (*fn:position()*) gesucht, kann auch dieses direkt über den Querykontext abgefragt werden, wie in *Abblidung 7* gezeigt.

```
1       switch (func)  {
2           case POS:
3               return Integer.get(contex.pos);
4           case LAST:
5               return Integer.get(contex.size);
```

Abbildung 7: XQuery Funktionen *fn:last()* und *fn:position()*

Auch das Auswerten der *sequence*-Funktion scheint im ersten Moment wieder simpel zu sein. Das gleiche Vorgehen wie bei den *Aggregate*-Funktionen wird auch tatsächlich bei *index-of()*, *insert-before()* und *remove()* genutzt. Es wird bis zum gesuchten Item iteriert und als Ergebnis ausgegeben. Jedoch gibt es ein intelligenteres Vorgehen im Bezug auf die Funktion *distinct-values()*.

fn:distinct-values() sucht aus einer oder mehreren Mengen von Items die verschiedenen Elemente heraus und verwirft den Rest – dadurch wird sie duplikatfrei. Auf den ersten Blick scheint eine vorherige Sortierung der Daten ein Muss zu sein. Danach können Duplikate entfernt werden, indem immer das letzte Item im Speicher gehalten und mit dem Neuen verglichen wird. Jedoch lässt sich die Funktion auch effizienter und vor allem ohne vorherige Sortierung implementieren.

Mit der Verwendung einer HashMap kann man einen sehr guten Kompromiss zwischen der *Nicht-Sortierung* und dem *„im-Speicher-behalten“* schließen. So können nicht sortierte Items einfach durch die *next()*-Methode in der HashMap indiziert werden und trotzdem das iterative Konzept an sich beibehalten werden (siehe *Abblidung 8*). Jedoch wird dadurch der Vorteil der Speicherkonstante teilweise gebrochen. In Anbetracht der Nichtsortierung scheint dies eine gute Alternative zu sein.

In BaseX wurde speziell für diese Funktion eine eigene HashMap-Klasse erstellt (*Abb. 8 – Zeile 6*), die auf deren Bedürfnisse zugeschnitten ist. So wird ein bool’scher Wert bei der Indizierung zurückgegeben (Abb. 8 – Zeile 15), der weitere Operationen in der Funktionsklasse selbst vermeidet.

```
1   private Iter distinct (final QueryContext ctx) throws QueryException
2   {
3       return new Iter()
4       {
5           final Iter eingabe_Iterator = expr[0].iter(ctx);
6           final ItemSet map = new ItemSet();
7
8           @Override
9           public Item next() throws QueryException
10          {
11              while(true)
12              {
13                  Item item = eingabe_Iterator.next();
14                  if (item == null) return null;
15                  if (map.index(item)) return item;
16              }
17          }
18      }
19  }
```

Abbildung 8: Vereinfachtes Codebeispiel von *fn:distinct-values()*

3.3.2 Path Expression - Location Path

Path Expression in XQuery sind XPath 2.0-Expression. Sie benutzen eine Pfadbezeichnung, um *Nodes* in einem XML-Dokument zu lokalisieren. Wie schon im *Kapitel 2.1 „Speicherung“* vorgestellt wurde, kann das XML-Dokument aber auch als Baumstruktur dargestellt werden. Eine *Path Expression* beschreibt dann also den Weg durch diesen Baum, um die gesuchten *Nodes* zu finden.

Eine *Path Expression* besteht aus einer Folge von einem oder mehreren *Steps*, wobei ein *Step* (oder auch *Location Step*) dabei entweder ein Achsenschritt, ein *Node Test oder* ein optionales Prädikat sein kann, welches dann als Filter fungiert.

Die „Pfadbeschreitung“ startet bei einer Sequenz von *Context Nodes*, die zu Beginn gesetzt wird. Durch jeden Schritt (bzw. *Step*) wird diese Sequenz durch eine neue ersetzt, welche aus dem Ergebnis der Sequenz von *Context Nodes* und dem *Step* besteht (*Abbildung 9*) [16].

scn0 / *step1* / *step2* / *step3*
scn1 / *step2* / *step3*
scn2 / *step3*

Sequenz von *context nodes:* *scn*

Abbildung 9: Multiple *Step* XPath-Pfad

Das Ergebnis einer *Path Expression* kann eine große Menge an *Nodes* aus einem XML-Dokument beinhalten, wie schon in *Kapitel 1.2 „Iterative Anfrageverarbeitung“* angedeutet wurde. Beispielsweise kann die Benutzung einer *Descendant-or-Self* Achse schnell dazu führen, dass alle Nodes eines Dokuments abgerufen werden. Deswegen ist die effiziente Implementation von XPath in BaseX einer der Hauptinteressen gewesen. Dazu gehört auch die iterative Umsetzung von möglichen Achsenschritten sowie *Filter Expression*.

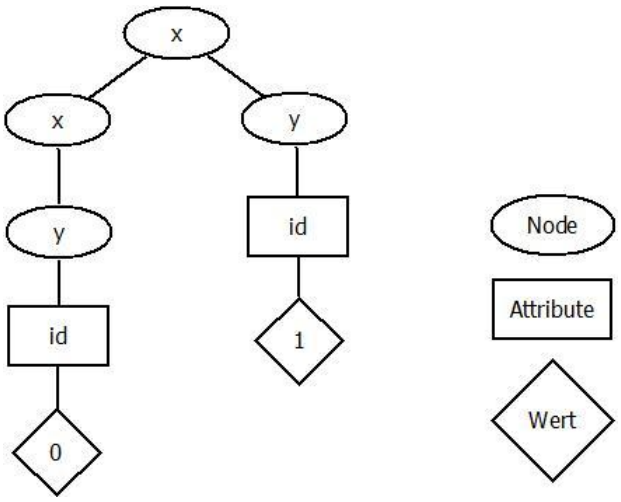
Das Vorhandensein von zwölf verschiedenen Achsenschritten, von denen sieben *Forward*-Achsen und fünf *Reverse*-Achsen sind, deren Kombination in einer Query fast beliebig sein können, macht die vollständige iterative Umsetzung für jeden individuellen Fall zu einem komplexen Sachverhalt. Zudem bestimmt die XPath Semantik [12], dass die resultierende Nodesequenz in *Document Order* und ohne Duplikate sein muss.

Zuerst wird nun ein einfaches konkretes Beispiel vorgeführt, um die Komplexität dieser Umsetzung zu veranschaulichen. Im weiteren Verlauf wird dann ein endlicher Automat beschrieben, der alle restlichen Fälle abdeckt.

3.3.2.1 Triviale Location Paths

Die Navigation durch den Baum und die Eigenschaft der Ordnung soll durch ein einfaches Beispieldokument mit simplen Anfragen verdeutlicht werden.

```
Beispiel.xml:
<x>
    <x> <y  id="0" /> </x>
    <y  id="1" />
</x>
```



Pre	Dist	Size	Kind	Content
0	1	7	Doc	Beispiel.xml
1	1	6	ELEM	X
2	1	3	ELEM	X
3	1	2	ELEM	Y
4	1	1	ATTR	Id="0"
5	4	2	ELEM	Y
6	1	1	ATTR	Id="1"

Abbildung 10: Beispiel XML Dokument mit dazugehörigen Baum und Pre/Post Tabelle

Abbildung 10 besteht aus einem XML-Dokument, einen dazugehörigen XML-Baum und dessen pre/post-Tabelle, die im Allgemeinen schon in Kapitel *2.1 „Speicherung"* näher erklärt wurde.

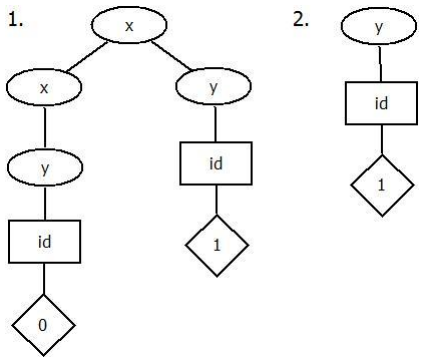
```
doc("beispiel.xml")/x/y
```

Abbildung 11: Beispielquery bestehend aus zwei *Child Steps*

Die erste Anfrage ist eine simple Query mit zwei *Child*-Achsen (*Abbildung 11*). Wie im vorherigen Kapitel gezeigt, wird durch dem ersten *Child Step* – Achsenschritt x – eine neue Sequenz von *Context Nodes* erzeugt. Diese ist in *Abbildung 12* mit *„1."* gekennzeichnet. Da es ein simpler *Child*-Schritt ist,

kann dadurch die Ergebnismenge nicht in einen ungeordneten Zustand gelangen. Ausgehend von einem Knoten sind dessen Kinder, auch wenn im Dokument auf dieser Ebene mehrere Knoten mit gleichem Name vorkommen sollten, immer in *Document Order*. Dies wird bei einem Vergleich mit der pre/post-Tabelle klar. Denn bildlich gesprochen werden die Kinder im Baum von „links nach rechts“ ausgewertet, also in der pre/post-Tabelle von oben nach unten und daher mit ansteigendem Pre-Wert.

Auch im zweiten Schritt wird ausgehend von der Ergebnismenge ein *Child*-Schritt gemacht und wieder gelangt man in einen geordneten Zustand, der mit *„2.“* gekennzeichnet ist.



Pre	Kind	Content
1	ELEM	X
5	ELEM	Y

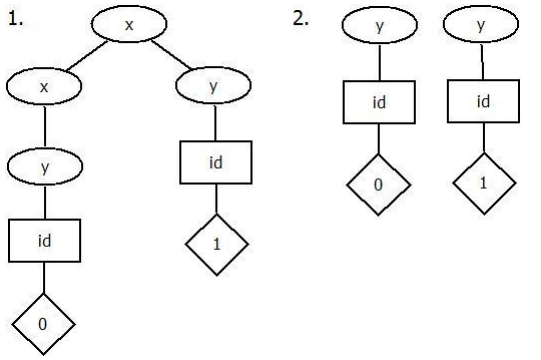
Abbildung 12: Ergebnis der Beispiel Query aus *Abbildung 11*

Bezogen auf das nächste Kapitel (*Kapitel 3.3.2.2 „Order Automat“*) kann gesagt werden, dass Queries mit beliebig vielen *Child Steps* immer in einem geordneten Zustand bleiben und sogar immer duplikatfrei sind.

```
doc(“beispiel.xml“)/x//y
```

Abbildung 13: Beispielquery bestehend aus *Child und Descendant-or-Self Step*

Die zweite Beispielanfrage besteht aus einer *Child*- und einer *Descendat-or-Self*-Achse (*Abbildung 13)*. Der erste Schritt resultiert im selben Ergebnis wie die vorherige Anfrage. Mit dem zweiten Schritt werden beide *y-Knoten* als Resultat erkannt, jedoch in richtiger Reihenfolge, wie in *„2.“* dargestellt in *Abbildung 14*, denn wie aus der pre/post-Tabelle zu entnehmen ist, hat der Knoten *„y“* mit der *„id=‘1‘“* einen höheren Pre-Wert und damit eine spätere Position in der D*ocument Order*.



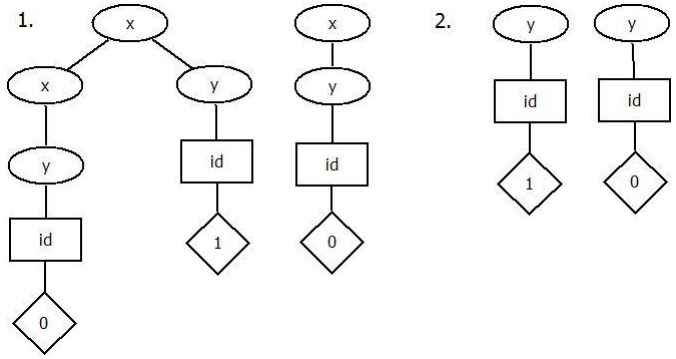
Pre	Kind	Content
1	ELEM	X
3	ELEM	Y
4	ATTR	Id=“0“
5	ELEM	Y
6	ATTR	Id=“1“

Abbildung 14: Ergebnis der Beispiel Query aus *Abbildung 13*

Die letzte Anfrage ist die „Problemquery“ (*Abbildung 15*) [16]. Auf einen *Descendant-or-Self Step* folgt ein *Child Step*.

```
doc(“beispiel.xml“)//x/y
```

Abbildung 15: Beispielquery bestehend aus *Descendant-or-Self Step* mit *Child Step (Quelle [16])*



Pre	Kind	Content
1	ELEM	X
2	ELEM	X
3	ELEM	Y
4	ATTR	Id=“0“
5	ELEM	Y
6	ATTR	Id=“1“

Abbildung 16: Ergebnis der Beispiel Query aus *Abbildung 15*

Der erste Schritt resultiert in einer Sequenz von zwei Knotenmengen (*Abbildung 16*). Diese selbst sind noch in einem geordneten Zustand, da der *x-Knoten* mit dem Pre-Wert *„1“* vor dem x-Knoten mit Pre-Wert *„2“* erkannt wird. Doch durch den anschließenden *Child Step* wird der *y-Knoten* mit *„id=‘1‘“* vor dem *y-Knoten* mit der *„id=‘0‘“* extrahiert. Somit wird laut pre/post-Tabelle ein Sortierschritt fällig, der diese beiden Knoten wieder ordnet.

Dieser kleine Auszug an Beispielen lässt erahnen, dass es eine große Menge an verschiedenen Kombinationsmöglichkeiten bei zwölf Achsen gibt, welche im nächsten Kapitel noch eingehender betrachtet werden.

3.3.2.2 Order Automata

Der *Order*-Automat [10] (siehe *Abbildung 17*) ist ein deterministischer Automat der, laut Entwickler, alle möglichen Kombinationen von Location Paths abdeckt und dabei angibt, ob sich der aktuelle Zustand in *Document Order* befindet, wobei *Self Step* und *Attribute Step* nicht mit einbezogen werden.

Der in *Kapitel 3.3.2.3 „No Duplicates Automata"* beschriebene Automat und dieser *Order*-Automat werden von 65 Inferenz-Regeln abgeleitet. Durch diese Regelmenge werden auch die Eigenschaften innerhalb eines Zustandes beschrieben. Regeln, sowie eingehende Erklärung der Eigenschaften sind im Paper [10] zu finden. Die wichtigsten Eigenschaften kurz im Überblick:

- ord_m
  ord_0 zeigt, dass der Zustand sich in *Document Order* befindet. Wobei der Index *m* angibt, wie viele Zustände es von der *Document Order* (dem möglichen Endzustand) entfernt ist. Der Zustand kann Duplikate enthalten.

- *nodup*
  Der Zustand hat keine Duplikate kann aber in einem ungeordneten Zustand sein (siehe *Kapitel 3.3.2.3 „No Duplicates Automat"*).

- *gen*
  Alle Ergebnisknoten des Zustandes haben den gleichen *dist*-Wert (Distanzwert; siehe *Kapitel 2.1 „Speicherung"*) zum Wurzelknoten.

- *unrel*
  Es gibt keine zwei verschiedenen Ergebnisknoten des Zustandes, die eine *Ancestor-Descendant* Beziehung haben.
  Beispiel: n_1 und n_2 sind Ergebnisknoten eines Zustandes. Dann kann n_1 kein „*Ancestor*-Knoten" von n_2 sein noch n_2 ein „*Ancestor*-Knoten" von n_1.

- lin_m
  Gibt an, ob es eine lineare Beziehung zum *Ancestor*-Knoten gibt, wobei der Index *m* angibt, wie viele Zustände dieser von dem möglichen Endzustand entfernt liegt

- *max1*
  Der Zustand hat maximal ein Ergebnis.

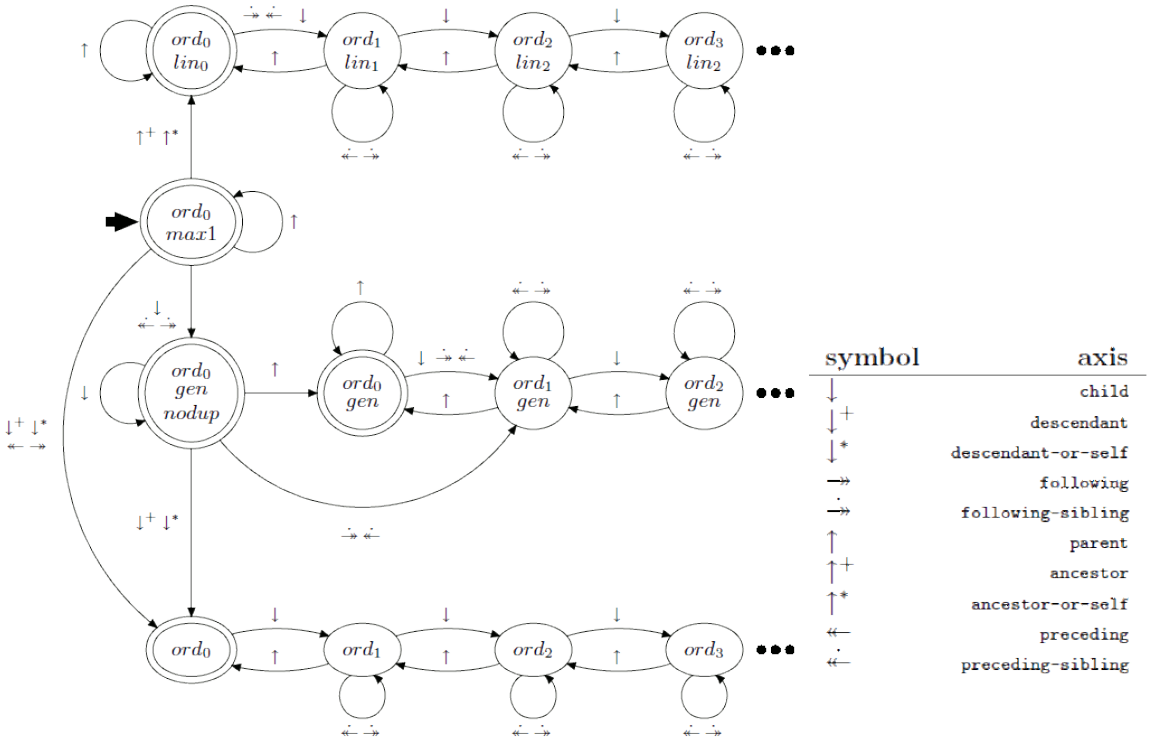


Abbildung 17: Order Automat (Quelle [10])

In *Abbildung 17* sind nur Übergänge angegeben, von denen aus ein geordneter Zustand erreicht werden kann. Alle sogenannten *Sink States* wurden übersichtshalber weggelassen, können aber in [10, *Appendix B*] nachgeschlagen werden. Denn sobald ein *Sink State* einmal erreicht wurde, ist es nicht mehr möglich, in einen geordneten Zustand zu gelangen.

Der Algorithmus, der hieraus abgeleitet werden kann, hat direkten Einfluss auf die Evaluationszeit einer Query. Werden bei einer relativen *Path Expression* wie zum Beispiel

*desc-or-self::b/c/foll-sibl::d/parent::**

sonst minimal fünf Sortierschritte fällig, so könnten durch den Einsatz des Algorithmus diese fünf Schritte weggelassen werden.

Durch den Automaten ist es zum einen möglich, die ganze Anfrage auf Sortiertheit zu überprüfen; zum anderen kann er auch verwendet werden, um Teilpfade bis zum Punkt des *Sink States* zu verfolgen und das Teilergebnis durch sortieren in *Document Order* zu bringen. Die ord_0 Eigenschaft ist wieder hergestellt und der Algorithmus kann von neuem angewendet werden.

Zudem lässt sich das Wissen der Ordnung auch insofern gut anwenden, dass geordnete Pfade in linearer Zeit von Duplikaten befreit werden können. Die Pipeline ist dadurch zwar unterbrochen, jedoch kann unter Umständen auch ein Performancevorteil geschaffen werden.

3.3.2.3 No Duplicates Automata

Der in *Abbildung 18* vorgestellte *No Duplicates*-Automat hat dieselben Eigenschaften und Inferenz-Regeln wie der *Order*-Automat [10]. Er stellt eine komplexe Suche nach Pfaden ohne Duplikate dar. Auch hier ist wieder ein erweiterter Automat mit allen *Sink States* in [10, *Appendix B*] zu finden.

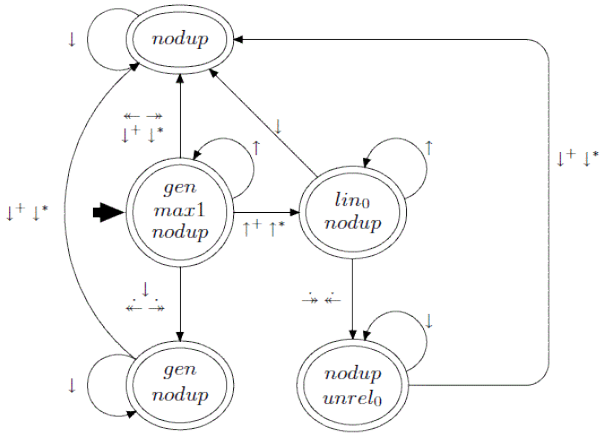


Abbildung 18: No Duplicates Automat (Quelle [10])

Auffällig bei diesem endlichen Automaten ist, dass sobald die *nodup*-Eigenschaft verloren ist, diese nicht wiederhergestellt werden kann. Im Gegensatz dazu konnte beim *Order*-Automat ein ungeordneter Zustand zurück zu einem geordneten führen.

Eine Symbiose der beiden Automaten lässt erkennen, dass zum Beispiel eine Query nur mit *Child*-Schritten sortiert und duplikatfrei ist. Dieses Wissen lässt sich daher sehr gut für das Pipelining verwenden und kann direkt als Algorithmus implementiert werden, da diese Queries keine weitere Bearbeitung benötigen.

3.3.3 Path Expression - Path Summary

Path Summaries werden im *Query Optimizer* dazu genutzt, den Zugriff auf Path Expression zu verbessern. Dabei werden gleichnamige Kinderknoten zusammengelegt [9], wie in *Abbildung 19* dargestellt.

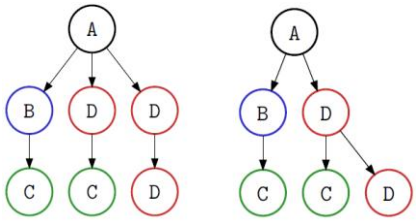


Abbildung 19: Beispiel für Path Summary (Quelle [21])

Aus dem linken breiter gefächerten Baum resultiert der rechte schlankere Baum. Er besitzt keine Attributwerte mehr, da er eine Indexfunktion für Path Expression hat. Aber zusätzlich kann durch die Zusammenfassung der Knoten der Zugriff auf die *Path Expression* effizienter gemacht werden, indem der *Location Path* umgeschrieben wird [21].

Der *Descendant Step* ist ein rechenintensiver Schritt, da alle Folgeknoten vom Wurzelknoten durchgegangen werden müssen. Wenn diese Expression nun durch mehrere *Child Steps* ersetzt wird, kann die Anzahl der kontaktierten Knoten auf den von der *Path Summary* erstellten verschiedenen Knotenpfaden reduziert werden.

Beispielsweise kann die *Path Expression* **//B,** auf das Dokument in *Abbildung 19* angewendet, in **/A/B** optimiert werden, da der Knoten **B** nur einmal vorkommt. Soll nun **//C** optimiert werden, wird analysiert, auf welchen Level (Pfadlänge zum Knoten) alle **C**-Knoten vorkommen. Sind diese auf einem Level, kann die optimierte Query **/A/*/C** erstellt werden, wobei ***** eine *Wildcard* darstellt und als *Node Test* alle Elemente auf diesem Level repräsentiert.

So kann durch die Optimierung auf *Child Steps* zum einen ein Performance-Vorteil sowie eine Reduzierung des Speicherverbrauchs erzielt werden. Zum anderen lässt sich dadurch auch manche Queries iterativ bearbeiten, da ja in *Kapitel 3.3.2.1 „Triviale Location Paths“* gezeigt wurde, das die Ordnung nach einen *Descendant Step* verlorengehen kann.

3.3.4 Path Expression – Predicate Expression

Ein weiteres nützliches Feature der Path Expression sind Prädikate, auch *Predicate Expression* genannt. Sie erlauben eine akkuratere Eingrenzung der gesuchten Daten und führen so zu kleineren und nützlicheren Ergebnissen. Prädikate geben immer einen bool'schen Wert zurück. Beispielsweise bezieht sich das Prädikat (*price > 20*) der Query

doc("catalog.xml")/catalog/product[price > 20]

auf den *price*-Kindknoten von dem *product*-Knoten und wird als *wahr* evaluiert, wenn er größer als 20 ist. Prädikate an sich können immer iterativ abgearbeitet werden, da sie die Reihenfolge einer Input-Sequenz nicht verändern, sondern nur aussortieren, welches der eingegebenen Items ein Ergebnis ist und welches nicht. In *Abbildung 20* wird dies beispielhaft dargestellt. Bildlich gesprochen wird von links nach rechts evaluiert.

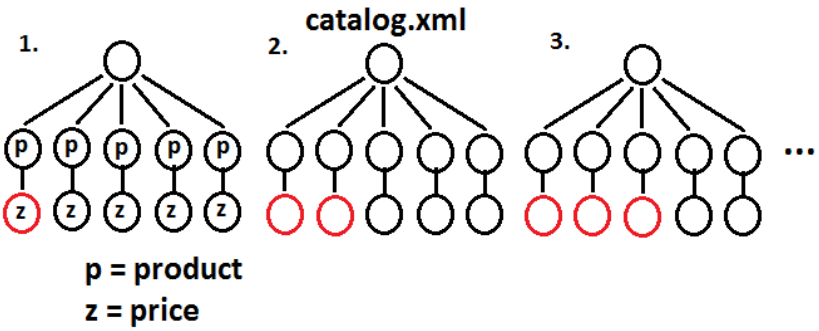


Abbildung 20: Beispiel für *Predicate Expression*

Eine Besonderheit stellen die Positionsprädikate dar. Wie schon in *Kapitel 3.3.1 "Primary Expression – Function Calls"* beschrieben, kann durch das Querycontext-Item die Größe des Iterators abgefragt werden. Daher kann auch hier bei den Positionsprädikaten sowie der *last()*-Funktion, die ja auch eine Positionsfunktion ist, der direkte Zugriff möglich sein. Sollte die Context-Item-Größe nicht bekannt sein, kann durch simple Iteration das gewünschte Item gefunden werden.

Dies stellt einer der Vorteile des Pipelinings dar. Denn angenommen das Dokument *catalog.xml* hat 1.000 *product*-Knoten, aber es wird nur der erste gesucht, so lässt sich mit der Query

doc("catalog.xml")/catalog/product[1]

die Suche nach dem Auffinden des ersten *product*-Knoten abbrechen. Mit der materialisierenden Methode müssten hier die gesamten 1000 *product*-Knoten in den Speicher geladen werden.

3.3.5 Sequence Expression

Filter Expression und *"Mengen"-Expression,* wie zum Beispiel *Union*, *Intersect* und *Except*, gehören zu dieser Kategorie der Expression. *Filter Expression* sind *Primary Expression* gefolgt von möglichen Prädikaten [2] und wurden daher schon im *Kapitel 3.3.4 "Path Expression - Predicates"* ausführlich behandelt.

Bei den *"Mengen"-Expression* soll die *Union Expression* als Stellvertreter dienen, da alle drei Operatoren nach dem gleichen Prinzip funktionieren. Sollen zwei Sequenzmengen mit dem *Union*-Operator durch eine iterative Methode zusammengefügt werden, sind hier wieder die zwei Grundvoraussetzungen der *Document Order* und Duplikatfreiheit von großer Wichtigkeit, die auch von der XQuery Semantik [12] verlangt werden. In *Abbildung 21* ist die Vorgehensweise der Methode schematisch dargestellt.

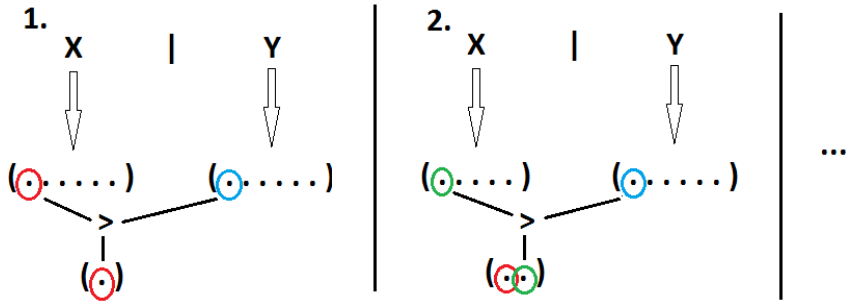


Abbildung 21: Beispiel für *Union Expression*

Es werden immer jeweils die ersten Items der zwei Menge verglichen und dabei das nach der *Document Order* „früher" vorkommende Item als nächstes Ergebnis weitergegeben. Daher ist das Endergebnis der *Union, Intersect* und *Except Expression* immer sortiert, was auch von nachfolgenden Expression genutzt werden kann. Angenommen zwei zu vergleichende Items sind gleich, dann wird eines der Items als Ergebnis gewertet und eines verworfen, da sonst Duplikate entstehen könnten.

Bei der *Intersect Expression* werden anstatt der unterschiedlichen nur die gleichen Items als Ergebnis gewertet, da ja die Schnittmenge gesucht ist - jedoch ist auch hier nur ein Item ein Ergebnis. Und entsprechend werden bei der *Except Expression* alle Items aus der ersten Menge abzüglich der gleichen Items aus der zweite Menge zurückgegeben.

4 Benchmarkergebnisse

Um die in dieser Arbeit theoretisch vorgestellten Methoden mit Ergebnissen zu belegen, wird als erstes eine auf das Pipelining zugeschnittene Query untersucht. Hierbei werden einerseits die Speichernutzung sowie die Performance in Betracht gezogen. *Abbildung 22* stellt die Resultate der folgenden drei Queries vor, die an sich keinen praktischen Nutzen haben, aber zur Veranschaulichung gut geeignet sind:

index-of (for $i in 1 to 100000 return 1, 1)
index-of (for $i in 1 to 1000000 return 1, 1)
index-of (for $i in 1 to 10000000 return 1, 1)

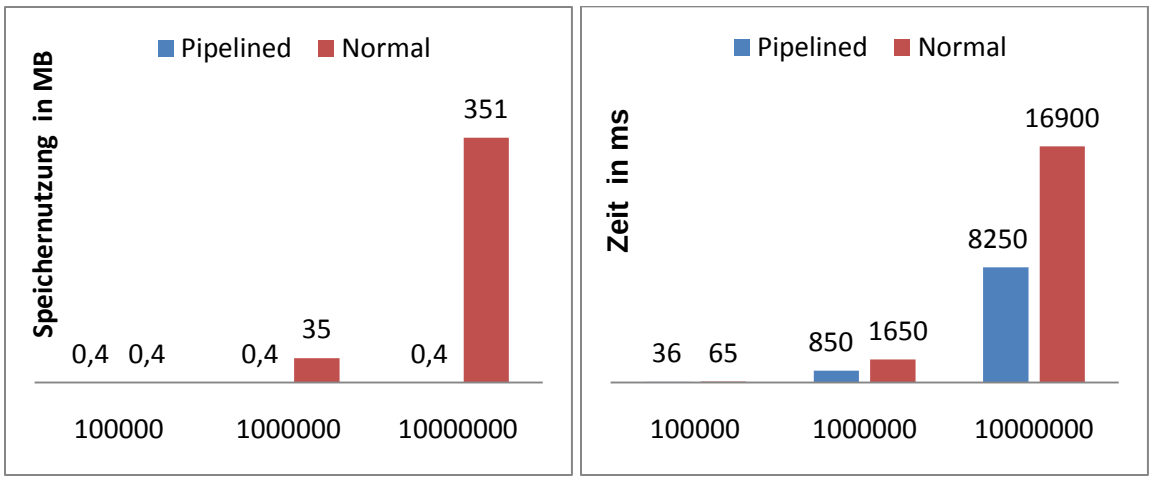


Abbildung 22: Benchmarkergebnisse Teil 1

Diese eindeutigen Ergebnisse zeigen, dass der konstante Speicherbedarf tatsächlich eingehalten wird. Bei einer Steigerung von 100 000 auf 10 000 000 Items als Ergebnismenge, bleibt die Speicherbelegung konstant bei 0,4 MB, während gleichzeitig bei der herkömmlichen Methode ein Anstieg der Speicherbelegung von 0,4 MB auf 351 MB stattfand. Durch die geringere Speicherbelastung ist die Performance bei der letzten Query über 50% besser.

Für das Benchmarking wurde die zurzeit aktuelle Version von BaseX (Version 6.28, Stand 11.09.2010) herangezogen. Die interne *Performance*-Klasse lässt viele Möglichkeiten zum Testen der Zeit, sowie auch des belegten Speichers. Auch die Möglichkeit der mehrfachen Wiederholung machen die Messergebnisse genauer.

Daher sind die hier präsentierten Ergebnisse relativ genau und sollten auch auf anderen Systemen reproduzierbar sein. Die absoluten Performancemessungen hängen natürlich vom System selbst ab, nicht jedoch die Speicherbelegung, die in dieser Arbeit ja eine wichtige Rolle spielt.

Da BaseX in Java programmiert ist, ist beim Auslesen des Speichers zu beachten, dass die *Java Garbage Collection* von der *Java Virtual Machine* gesteuert wird und daher automatisch aufgerufen wird. Da das iterative Prinzip darauf beruht, dass nicht mehr benötige Items verworfen werden können, muss für das Auslesen, die *Java Garbage Collection* manuell ausgeführt werden. Sonst kann nicht garantiert werden, dass tatsächlich auch nur die zurzeit genutzten Objekte im Speicher sind. So können auch schon verworfene Items noch im Speicher residieren.

Im nächsten Beispiel werden zwar praktischere, aber immer noch simple Query benutzt wurde. Sie wurden auf drei verschieden großen Datenbanken getestet (10 MB, 100 MB, 1000 MB), die jeweils mit dem *XMark Benchmark* [5] erstellt wurden:

doc('XmarkDatabase')/site/regions/namerica/item

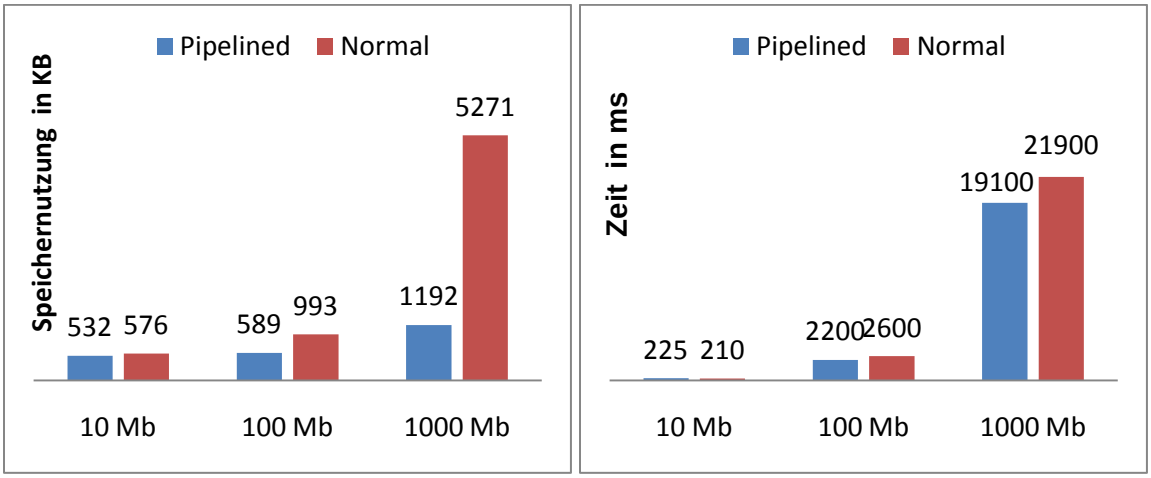


Abbildung 23: Benchmarkergebnisse Teil 2

Wie in *Kapitel 3.3.2.2 „Order Automata "* und *Kapitel 3.3.2.3 „No Duplicates Automata"* gezeigt, kann eine Query, die nur aus *Child Steps* besteht, komplett iterativ bearbeitet werden. Dennoch sind die in *Abbildung 32* dargestellten Ergebnisse der Query nicht mehr so eindeutig wie die Vorherigen. In der *10 MB*/*100 MB*/*1000 MB* Datenbank wurden 680/6800/68 000 Items als Resultat gefunden.

Bei diesem Beispiel ist wieder ein deutlicher Anstieg der Speicherbelegung bei der ursprünglichen Methode zu erkennen, jedoch ist auch ein leichter Anstieg bei der Pipelining-Methode zu beobachten. Dies lässt sich auf die Größe der Datenbank zurückführen: die Speichermessung kann nicht nur auf die durch den Iterator verursachte Speicherbelegung zurückgeführt werden, sondern auch Metadaten sowie für den Index benötigten Daten werden im Speicher gehalten.

Trotzdem wurde die Speicherbelegung auf der 1000 MB-Datenbank um mehr als 75% reduziert. Dies ist ein herausragendes Ergebnis, wenn es nicht als isolierte Anfrage gesehen wird, sondern als eine von tausenden Anfragen, die ein Server täglich bewältigen muss. Zu Letzt wird noch die *XMark Query 19* [5] vorgestellt, die als *„Real World“*-Beispiel betrachtet werden kann:

```
let $auction := doc(“XmarkDatabase") return
for $b in $auction/site/regions//item
let $k := $b/name/text()
order by zero-or-one($b/location) ascending empty greatest
return <item name="{$k}">{$b/location/text()}</item>
```

Sie erstellt eine alphabetisch geordnete Liste aller *Items* mit deren zugehörigen *Locations*.

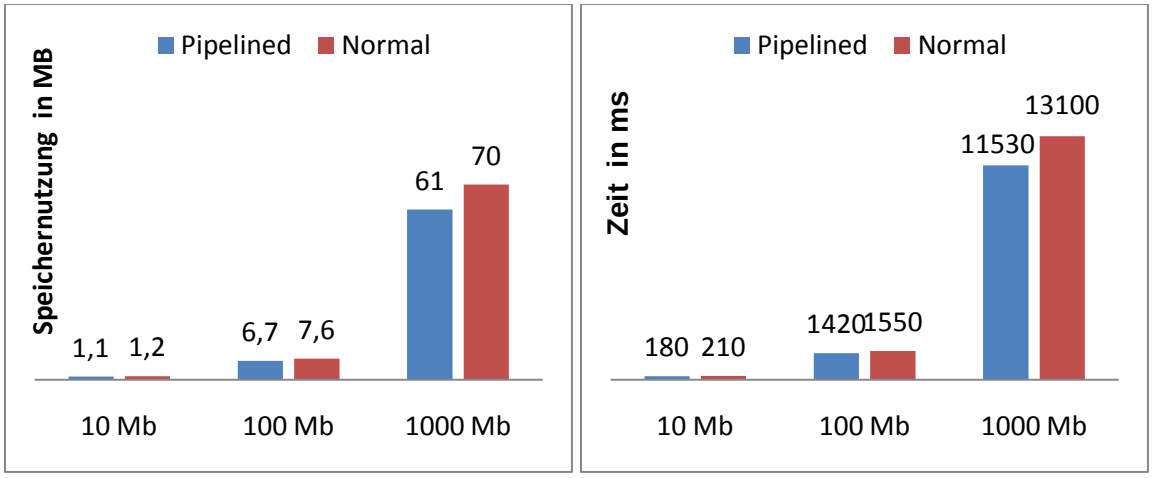


Abbildung 24: Benchmarkergebnisse Teil 3

Die Ergebnisse in *Abbildung 24* sind auf keinen Fall vergleichbar mit externen Tests, da für den Nicht-Pipeline Test nur die *Path Expression* umgeschrieben wurden und alle anderen Methoden unverändert blieben. Dennoch zeigt sich hier, dass allein diese Änderung Auswirkungen im einstelligen Prozent-Bereich hat. Doch grundsätzlich ist auch nicht zu erwarten, dass die komplette Query iterativ verarbeitet werden kann – allein deshalb schon, weil ein *OrderBy Statement* enthalten ist. *XMark Query 19* steht stellvertretend für alle *XMark Queries*, obwohl die Resultate oft nicht so gut ausfielen wie bei dieser Query. Dennoch war meistens eine positive Tendenz zugunsten des Pipelinings zu beobachten.

5 Relevante Forschungsarbeiten

Die Abfrage von persistenten XML-Dokumenten an Datenbanken ist nicht das einzige Forschungsfeld, in der das Pipelining relevant ist. Noch wichtiger ist es bei dem Datenmodel der kontinuierlichen Datenströme. Diese können zum Beispiel von fortlaufenden Datensammlern oder lang andauernden Computer-Simulationen kommen. Die Daten müssen in *real-time* analysiert werden und können dabei oft nur einmal abgefragt werden, da Daten schneller übermittelt werden, als sie in verteilten Systemen gespeichert und wieder abgerufen werden können.

Es wurde viel Aufwand für die Forschung der Evaluation von *XPath Expression* in kontinuierlichen Datenströmen betrieben. Durch die Ordnung von XML sind automatenbasierte Annäherungen eine Möglichkeit. Verschiedene Ideen endlicher Automaten für baumgleiche Strukturen wurden für das Streaming vorgeschlagen und auch umgesetzt. *Tree Walking Automata* sind ein traditionelles Beispiel [19] aus dem Jahre 1971, das aber nicht alle regulären Sprachen erfasste, denn was fehlte war ein *Stack*.

Hier führten 2004 Alur und Madhusudan [20] im Kontext der Programmverfizierung den *Visiby Pushdown Automata* (VPA) ein. Erst 2007 wurde die Relevanz eines *VPAs* für das XML-Streaming wieder herausgestellt [14]. Es basiert nicht auf einer Baumstruktur, denn laut Entwickler gehört das Verarbeiten von *Streams* großer XML-Dokumente (zum Beispiel gestreamte Böresenkursdaten) zu der Klasse, für die dies nicht möglich ist. Es lohnt sich nicht, XML-Dokument in einen Baum zu konvertieren und dann erst abzufragen, weil *Streaming Algorithmen* Daten verarbeiten müssen, sobald sie eintreffen [14]. Dies muss schnell und mit möglichst geringem Aufwand vonstattengehen.

Ein *VPA* ist ein *Pushdown Automata*, dessen Stackeinsatz vom Input den er liest festgelegt wird. Ein XML-Dokument ist am besten als ein *Nested Word* anzusehen, eine lineare Struktur (Wort) in einer geschachtelten Beziehung, die als *Open Tags* und deren entsprechenden *Close Tags* dargestellt werden (Farben entsprechen *Abbildung 24)*. *VPAs* können also die geschachtelte Relation rekonstruieren, indem sie einen *Open Tag* auf den Stack legen und durch den entsprechenden *Close Tag* wieder aus den Stack entfernen [17,18].

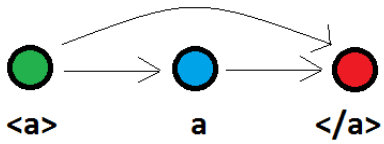
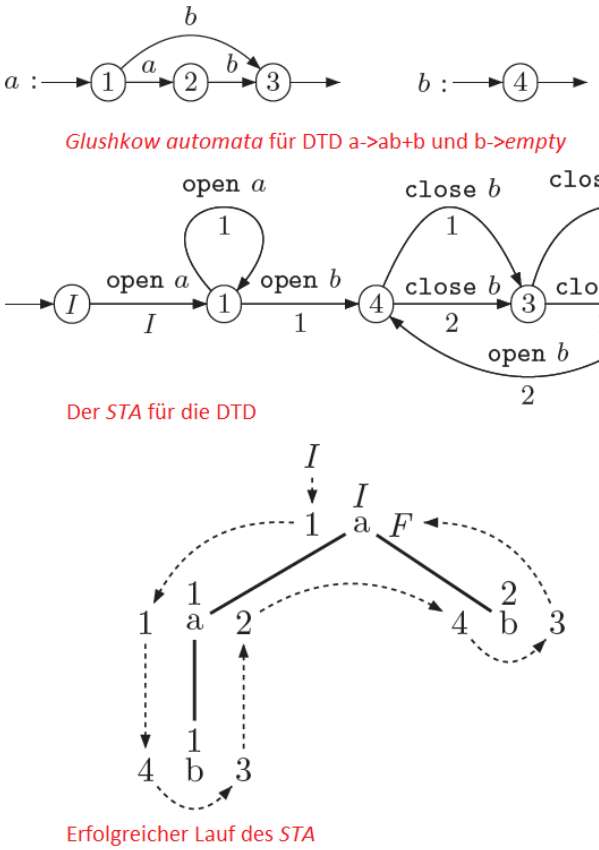


Abbildung 25: Vorgehensweise eines *Visibly Pushdown Automata*

Sie sind genauso leistungsfähig wie reguläre Sprachen, die die Baumstruktur akzeptieren. Doch ihre wichtigste Eigenschaft ist, dass sie entscheidbar (“*determinizable*“) sind [14,25].

Die vorher erwähnten *Tree Walking Automata,* wurden 1998 von Neumann und Seidl durch den *Pushdown Forest Automata (PFA)* fortgesetzt und für die *XML Query Evaluation* vergeschlagen. Auch hier wurde 2007 die Gemeinsamkeiten des *PFAs* und *VPAs* registriert [14] und von Gauwin und Roos 2008 in *Streaming Tree Automata (STA)* [17] zusammengelegt. Dieser agiert nun direkt auf Bäume mit unbeschränkten Verzweigungsgrad (*unranked trees*). Dies sei, laut Entwickler, der „natürlichste *Tree Automata* für alle möglichen XML-Streaming-Anwendungen“ [17].



Glushkow automata für DTD a->ab+b und b->*empty*

Der *STA* für die DTD

Erfolgreicher Lauf des *STA*

Ist eine deterministische DTD gegeben, wird die Menge der *Glushkov Automata* berechnet, welche deterministische endliche Automaten für die *Regular Expression* der DTD sind.

Aus dieser Menge wird ein deterministischer *STA* konstruiert. Dabei werden die neuen Zustände *I* und *F*, als Start- und Endzusatand hinzugefügt, sowie die Menge der Glushkov Automaten vereinigt.

Abbildung 26: Vorgehensweise eines Streaming Tree Automata (Quelle [17])

Beide Automaten, der *VPA* und *STA*, sind laut Entwickler gut für das Streaming geeignet, können jedoch noch nicht alle *XQuery Expression* streamen. Da sie eine ganz unterschiedliche Voraussetzung haben als bei dem Abfragen von persistenten Daten, ist dieses Forschungsfeld vermutlich für BaseX uninteressant. Dennoch lässt sich vielleicht aus der hier vorgestellten Vorgehensweise ein abstrakteres Problem im Zusammenhang mit dem Pipelining definieren. Eventuell werden die beschriebenen Automaten unter anderen, noch nicht absehbaren Umständen für XML-Datenbanken interessant.

6 Zusammenfassung

In dieser Arbeit wurde das Konzept der iterativen Anfrageverarbeitung sowohl theoretisch beleuchtet als auch die praktische Umsetzung in BaseX angedeutet.

Bei diesem Konzept sind die zwei Voraussetzungen der Sortiertheit sowie der Duplikatfreiheit ein essentieller Bestandteil der ankommenden Daten. Daher wurde im zweiten Kapitel herausgestellt, wie die *Document Order* (Sortiertheit) direkt mit dem physikalischen Speichermodel zusammenhängt. Die genutzte Variante des p*re/post*-Encodings speichert ***pre/dist/size***-Triplets der XML-Knoten ab, wobei der p*re*-Wert dem Platz in der *Document Order* entspricht.

Während bei dem materialisierenden Model in der Funktion selbst das Ergebnis komplett ausgewertet wird, wertet der iterative Ansatz ein Ergebnis "*just-in-time*" aus. Dabei ergeben sich Vorteile, wie zum Beispiel der theoretisch konstante Speicherbedarf, da immer nur ein Item im Speicher ist ein Performancegewinn, da Anfragen, die zum Beispiel nur die ersten 100 von tausenden Ergebnissen benötigen, einfach abgebrochen werden können ohne das alles ausgewertet werden muss. Dem gegenüber steht die gesteigerte Entwicklungszeit der Software, da zum einen immer noch die materialisierende Methode implementiert wird und zum anderen die Implementation an sich komplexer ist.

Im dritten Kapitel wurden konkrete Umsetzungen in BaseX sowie einige theoretische Ansätze aufgezeigt. Die hier gezeigten *Primary Expression* sind an sich einfachere Umsetzungen, bei denen es sich größtenteils um *Function Calls* handelt. Dagegen beschäftigen sich viele veröffentlichte wissenschaftliche Paper mit der iterativen Bearbeitung von *Path Expression*, die eine deutlich komplexere Struktur besitzen. Mit der relativ trivialen Beispielquery

doc("beispiel.xml")//x/y

wurde gezeigt, warum hier die *Document Order* ohne nachträgliches Sortieren nicht mehr vorhanden ist. Durch den vorgestellten *Order Automata* konnte im Zusammenhang mit dem *No Duplicates Automata* einige *Path Expression* gefunden werden, die immer iterativ arbeiten können, da sie keine nachträgliche Bearbeitung benötigen. Durch die Optimierung des *Path Summary* wurde zudem der aufwändig *Descendat-or-self Step* in multiple *Child Steps* konvertiert, was auch dem Pipelining zuträglich ist.

Die Benchmarkergebnisse haben die Vorteile des Pipelinings belegt und haben gezeigt, dass sowohl die Speicherbelegung, als auch die Performance sich durch diese Maßnahme verbesserten.

Literatur

[1] W3C Dom Working Group. Document Object Model (DOM). http://www.w3.org/DOM/, 2004.

[2] XQuery Group. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, W3C Recommendation 23 January 2007.

[3] C. Grün et al. BaseX - Processing and Visualizing XML with a native XML Database, http://www.basex.org/, 2010.

[4] G. Graefe. Query Evaluation Techniques for Large Databases. ACM Computing Surveys, 25(2): 73-170, 1993.

[5] CWI, Niederlande. XMark – an XML Benchmark Project, http://www.xml-benchmark.org/, 2009.

[6] C. Grün et al. XQuery Full Text Implementation in BaseX, 2009.

[7] T. Grust et al. MonetDB/XQuery – Consistent & Efficient Updates on the Pre/Post Plane, 2006.

[8] P. Boncz, S. Manegold, J. Rittinger. Updating the Pre/Post Plane in MonetDB/XQuery, 2005.

[9] R. Goldmann, J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases, 1997.

[10] J. Hidders, P. Michiels. Avoiding Unnecessary Ordering Operations in XPath, 2003.

[11] XQuery Group. XQuery 1.0 and XPath 2.0 Functions and Operators. http://www.w3.org/TR/xquery-operators/, W3C Recommendation 23 January 2007.

[12] XQuery Group. XQuery 1.0 and XPath 2.0 Formal Semantics. http://www.w3.org/TR/xquery-semantics/, W3C Recommendation 23 January 2007.

[13] L. Fegaras, R. Dash, Y. Wang. A Fully Pipelined XQuery Processor, 2006.

[14] V.Kumar, P. Madhusudan, M. Viswanathan. Visibly Pushdown Automata for Streaming XML. In 16th international conference on World Wide Web, pages 1053-1062, ACM-Press, 2007.

[15] R. Alur. Nested Words aka Visbly Pushdown Languages. http://www.cis.upenn.edu/~alur/nw.html, Stand 01.09.2010.

[16] M. Scholl. Vorlesung: XML and Databases. Universität Konstanz, Winter 2007/2008.

[17] O. Gauwin, J. Niehren, Y. Roos. Streaming Tree Automata, 2008.

[18] R. Alur. Marrying Words and Trees, 2007.

[19] A. Aho, J. Ullmann. Translation on a context-free grammar. 19:439-475, 1971.

[20] R. Alur, P. Madhusudan. Visibly pushdown languages. In 36th ACM Symposium on Theory of Computing, pages 202-211. ACM-Press, 2004.

[21] C. Grün. PhD Thesis: Storing and Querying Large XML Instances. Nov. 2010.

Abbildungsverzeichnis