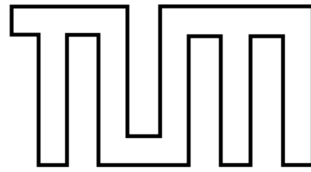


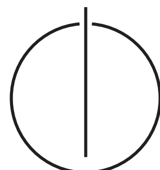
Fakultät für Informatik

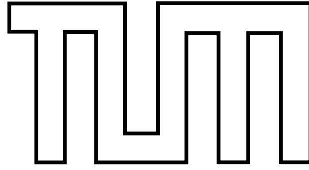
der Technischen Universität München

Diplomarbeit in Informatik

Ein kdb+ - Code-Generator für den Pathfinder XQuery Compiler

Thorsten Kandler





Fakultät für Informatik

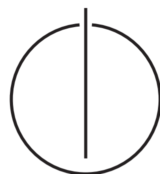
der Technischen Universität München

Diplomarbeit in Informatik

Ein kdb+ - Code-Generator für den Pathfinder XQuery Compiler

A kdb+ Back-End for the Pathfinder XQuery Compiler

Bearbeiter:	Thorsten Kandler
Aufgabensteller:	Prof. Dr. Torsten Grust
Betreuer:	Jan Rittinger, M.Sc.
Abgabedatum:	15. Januar 2008



Erklärung

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

15. Januar 2008

Zusammenfassung

Diese Diplomarbeit beschäftigt sich mit der Implementation eines kdb+-Code-Generators für den Pathfinder XQuery-Compiler. Es wird die Abbildung der XML/XQuery Welt auf die des relationalen Datenbanksystemes kdb+ dargestellt.
kdb+ bildet mit seinem speziellen Datenmodell der vertikalen Fragmentierung von Tabellen und seiner listenbasierten Programmiersprache q eine interessante Zielarchitektur für XQuery.
In dieser Arbeit werden Konzepte vorgestellt, um XQuery Anfragen durch kdb+ verarbeiten zu lassen.

Inhaltsverzeichnis

1 Einführung **1**

2 Relationale Algebra **4**
2.1 Tabelle 4
2.2 Attach 5
2.3 Tupelfunktionen 5
2.4 Selektion 6
2.5 Projektion 6
2.6 Aggregationen 6
2.7 Joins 7
2.8 Distinct 7
2.9 Mengenoperation 8
2.10 Kreuzprodukt 8
2.11 Tupelnummeriung 8
2.12 Typumwandlung 9
2.13 Dokumentensuche 9
2.14 Dokumenteneinsicht 9
2.15 XPath-Step 9
2.16 Konstruktoren 10

3 kdb+/q **13**
3.1 kdb+ 13
3.2 q 14

4 XML Daten in kdb+ **22**
4.1 Relationales Layout 22
4.2 Mehrfache Fragmente 25

5 Übersetzung **28**
5.1 Abarbeitungsplan 28
5.2 Folgerungsregel 29

5.2.1 Y-Operator 29
5.3 Tabelle 30
5.4 Attach 31
5.5 Tupelfunktionen 32
5.6 Selektion 33
5.7 Projektion 34
5.8 Aggregationen 34
5.9 Joins 35
5.9.1 Equi-Join 35
5.9.2 Semi-Join 37
5.9.3 Theta-Join 38
5.10 Distinct 40
5.11 Mengenoperationen 41
5.12 Kreuzprodukt 42
5.13 Tupelnummerierung 43
5.14 Typumwandlung 44
5.15 Dokumentensuche 45
5.16 Dokumenteneinsicht 46
5.17 XPath-Step 47
5.17.1 Implementierung von $\hat{\lrcorner}_\alpha$ 49
5.17.2 Descendant-Step 50
5.17.3 Descendant-Step optimiert 51
5.17.4 Child-Step 52
5.17.5 Child-Step optimiert 53
5.18 Konstruktoren 53
5.18.1 Textnode 54
5.18.2 Attribut 55
5.18.3 FCNS 55
5.18.4 Element 56
5.18.5 Content 59
5.18.6 Twig 64

6 Experimente **66**
6.1 Konfiguration 66
6.2 Equi-Join 67
6.3 Descendant-Step 68
6.4 Child-Step 68
6.5 Verschiedene Größen 69

7 Resümee **71**

A Prototyp pf2kdb **73**

B Beispielübersetzung **74**
B.1 Abfrage in XQuery 74
B.2 XML Ergebnis 74
B.3 Algebra Plan 75
B.4 q Code 76

C Laden von CSV Dateien **78**

D Beigefügte DVD **79**

Kapitel 1

Einführung

Pathfinder[1] ist ein XQuery Compiler für XQuery 1.0 [W3C07] Abfragen, die er in Abarbeitungspläne mit Operatoren der relationalen Algebra umwandelt. Die generierten Pläne sind so konzipiert, dass Übersetzungen in reell ausführbaren Code auf Datenbanksystemen möglich sind. Auf den Datenbanksystem müssen dazu relationale Darstellungen der XML Dokumente hintergelegt sein, auf denen der generierte Ziel-Code arbeiten kann. Durch Pathfinder gelingt es also, XQuery-Abfragen — auch als Abfragen über der Baumdarstellung eines XML Dokumentes zu sehen — auf relationale Anfragen über relationale Kodierungen der Baumdarstellungen umzusetzen. Das Ziel von Pathfinder ist es, über das Nutzen von etablierten und existierenden relationalen Datenbanksystemen beschleunigte Abfragenausführungen zu erlangen.

XQuery 1.0 ist eine präzise jedoch flexible Anfragesprache für XML. Es stellt das XML-Pendant zu SQL dar und es ist anzunehmen, dass es zusammen mit der wachsenden Nutzung von XML an Relevanz gewinnt. Zwar besitzen XQuery und SQL Ähnlichkeiten bezüglich Zweck und Syntax, es besteht aber der wesentliche Unterschied, dass SQL auf ungeordnete Mengen *flacher* Tupel basiert wohingegen sich XQuery an geordnete Mengen von Werten sowie an hierarchisch angeordnete Knoten richtet.
XQuery integriert XPath und dessen Pfadausdrücke zur Auffindung bestimmter Knoten im XML Dokument. Mit XQuery ist auch das Erstellen von XML-Strukturen durch Konstruktoren (Element, Text u.a.) möglich. Ein weiteres zentrales Element von XQuery sind die sogenannten *FLWOR*-Ausdrücke *(For,Let,Where,Order By,Return)*, die verschiedene Zwecke verfolgen: Iteration, Variablendefintion, Ergebnisfilterung und Sortierung [Bru04]. Pathfinder

[1] http://www.pathfinder-xquery.org

bietet eine spezielle Heransgehensweise, genannt Loop-Liftung, zur effizienten Handhabung von beispielsweise Pfad-Ausdrücken innerhalb von FLWOR-Ausdrücken. Es stützt sich auf die Tatsache, dass XQuery ohne Seiteneffekte agiert. Dabei stellt eine einzige Relation den Inhalt aller Iterationen eines FLWOR-Ausdruckes dar. Diese Relation ist entsprechend der Iterationen in Partitionen aufgeteilt (siehe dazu [BGvK+05] und [GST04]).

Bislang wurden im Rahmen des Pathfinder Projektes die Integration verschiedener Datenbanksysteme als Back-End für Pathfinder, wie MonetDB (MonetDB/XQuery) und IBM DB2, verwirklicht. Dabei wurde die relationale Algebra von Pathfinder in die spezifische Sprache MIL für MonetDB bzw. in SQL-1999 für DB2 übersetzt. Als weiterer Back-End soll nun kdb+ in Erscheinung treten.

Kdb+ ist ein relationales Datenbanksystem entwickelt von *Kx systems* ([kx]). Es ist ein beliebtes Werkzeug für Zeitreihenanalysen sehr großer Datenmengen. Typische Anwendungsfälle sind in [O'N06] beschrieben. kdb+ beinhaltet eine komplette Programmier- und Abfrage-Umgebung implementiert durch die Sprache q. q kann benutzt werden für Datenbankabfragen, Datenanalyse, Programmierung und Ereignisbehandlung und deckt einen weit größeren Bereich als Abfragesprachen typischer RDBMS ab, die ausschließlich für Datenabfragen benutzt werden können. Das Datenmodell der Skriptsprache q ist listenbasiert und Tabellen sind im Gegensatz zu üblichen RDBMS spaltenorientiert hinterlegt. Der Blickwinkel auf eine Relation oder Tabelle kann man sich dadurch als 90° gedreht vorstellen. Die Speicherung von Tabellenspalten als Listen und damit als Datensequenzen ermöglicht in kdb+ einen Zugriff anhand der Indexpositionen der Elemente. Besonders für XQuery mit dem darunterliegenden geordneten Datenmodell stellt kdb+ dadurch eine geeignete Plattform dar.

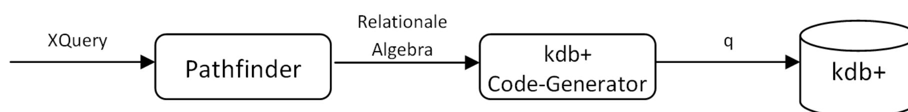


Abbildung 1.1: XQuery auf kdb+ mit Pathfinder

Ziel der Diplomarbeit ist kdb+ als Back-End für Pathfinder zu integrieren. Dies umfasst die Abbildung der relationalen Algebra-Operatoren wie sie von Pathfinder erzeugt werden auf die spezifische kdb+ Programmier- und Abfragesprache q. Als Ausgangspunkt für die Übersetzung zu kdb+, stellt die relationale Algebra von Pathfinder einen wichtigen Eckpfeiler zur Ein-

führung in die Thematik dar. In Kapitel 2 wird die relationale Algebra, wie sie von Pathfinder verwendet wird, vorgestellt. Die wesentlichen und relevanten Konzepte der Skriptsprache q bezüglich dieser relationalen Algebra werden in Kapitel 3 eingeführt, bevor in Kapitel 4 die Repräsentation von XML-Daten auf relationaler Basis vorgestellt wird. Kapitel 5 beschreibt korrespondierend zu Kapitel 2 die eigentliche Logik der Code-Generierung: die Übersetzung der einzelnen Operatoren. In Kapitel 6 wird die Übersetzung zu q in verschiedenen Varianten getestet und bemessen.

Kapitel 2

Relationale Algebra

Die relationale Algebra ist eine prozedural orientierte Sprache zur Extraktion von Informationen aus einer Datenbank. Sie beschreibt implizit in einem Abarbeitungsplan wie eine Abfrage ausgewertet wird (vgl. [KE06]). Sie umfasst fünf grundlegende Operationen: Selektion, Projektion, Kartesisches Produkt, Vereinigung und Mengendifferenz. Weiterführende Operationen wie Verbund (Join) und Schnittmenge können von den Basisoperationen abgeleitet werden. Die relationale Algebra besteht also aus Mengenoperationen wie auch aus Operationen, die spezifisch für relationale Datenbanksysteme entwickelt wurden, um Relationen von anderen abzuleiten [Cod70]. Die Fähigkeit Daten zusammenzufassen, ist als weiterer Bestandteil mit den Aggregatsfunktionen hinzugekommen.
Durch den Pathfinder XQuery-Compiler werden Abfragepläne basierend auf relationaler Algebra erstellt. Er adoptiert eine Vielzahl der Operationen der standardisierten relationalen Algebra. Um den Spagat von XQuery zur relationalen Welt möglichst effizient zu meistern, sind erweiternd zusätzliche Operatoren integriert.
In diesem Kapitel werden die benötigten Pathfinder Operatoren vorgestellt. Tabelle 2.1 gibt einen Überblick über die relevanten Operatoren für diese Arbeit.

2.1 Tabelle $\underline{a|b}$

Dieser Operator erstellt eine neue Basistabelle, bestehend aus einer oder mehreren Spalten sowie einer oder mehrerer Tupel. Die Werte dieser Tupel sind direkt im Operator als Information hinterlegt und werden nicht von Eingaberelationen bezogen.

$\underline{\mathsf{a}\vert\mathsf{b}}$	Tabelle mit Spalten a und b
$@_{\mathsf{a}:v}$	Attach von a mit konstantem Wert v
$\otimes$	Tupelfunktionen & Boole'sche Vergleiche
σ_{p}	Selektion auf Prädikatsattribut p
$\pi_{\mathsf{a}_1:\mathsf{b}_1,\ldots,\mathsf{a}_n:\mathsf{b}_n}$	Projektion
$sum, \ldots, max$	Aggregationen
$\bowtie_{\mathsf{a}=\mathsf{b}}, \bowtie_{\mathsf{a}\theta\mathsf{b}}, \ltimes_{\mathsf{a}=\mathsf{b}}$	Equi-Join, Semi-Join, Theta-Join (Joinkriterien a und b)
δ	Duplikateliminierung
$\setminus, \cap, \dot\cup$	Mengenoperationen
$\times$	Kreuzprodukt
$\rho_{\langle \mathsf{a}_1,\ldots,\mathsf{a}_n\rangle,\mathsf{b},\mathsf{p}}$	Tupelnummerierung
$cast_{type,n}$	Typumwandlung zu Datentyp *type*
$doc - lookup_{\mathsf{p},\mathsf{n}}$	Dokumentensuche
$doc - access_{\mathsf{p},\mathsf{n}}$	Dokumenteneinsicht
$⏍_{\alpha,n}$	*XPath-Step* (Achse α, Knotentest n)
Λ_t	Knotenkonstruktor des Types t

Tabelle 2.1: Operatoren der relationalen Algebra

2.2 Attach $@_{\mathsf{a}:v}$

$@_{a:v}$ erweitert die Eingaberelation um eine Spalte a. Alle Tupel der Ausgaberelation beinhalten in a den konstanten Wert v. $@_{a:v}$ ist äquivalent zu einem Kreuzprodukt der Eingaberelation mit einer Tabelle, die eine Spalte a mit einem Wert v besitzt.

$$\begin{array}{ccc} & \times & \\ \diagup & & \diagdown \\ \vdots & & \begin{array}{|c|}\mathsf{a}\\ \hline v\end{array} \end{array}$$

2.3 Tupelfunktionen & Boole'sche Vergleiche

Der Oberbegriff Tupelfunktionen beschreibt mehrere Funktionen, die auf Tupel-für-Tupel Basis operieren. Das umfasst arithmetische Operationen wie $+, -, *, /, mod$, Vergleichsoperationen wie $=, >$ sowie *built-in* Funktionen von XQuery wie $fn : ceiling$, $fn : round$ oder $fn : concat$. Abhängig vom Operator werden ein oder zwei Eingabespalten für die Funktion betrachtet. Die Ergebnisrelation wird jedoch immer um eine Spalte mit dem Ergebnis der Funktion erweitert. Ähnlich verhalten sich die Vergleichsoperationen *and*, *or* etc.
Die Übersetzung dieser Funktionen ist in Abschnitt 5.5 beschrieben.

Beispiel 2.1. *Betrachten wir nun die Tupelfunktion* $\mathsf{res}{=}\mathsf{item}_1{>}\mathsf{item}_2$ *auf einer Eingaberelation mit den Spalten* item_1 *und* item_2*. Die Funktionsausführung bewirkt eine tupelweise Addition der Elemente und legt das Ergebnis in der neuen Spalte* res *ab.*

(a) Eingabe-Relation

item_1	item_2
0	2
1	2
2	2
3	2

(b) Ergebnis von $\mathsf{res}{=}\mathsf{item}_1{+}\mathsf{item}_2$

item_1	item_2	res
0	2	*false*
1	2	*false*
2	2	*false*
3	2	*true*

Tabelle 2.2: Addition als Beispiel für Tupelfunktion

2.4 Selektion σ_{p}

Die relationale Algebra von Pathfinder vereinfacht Operatoren und drückt explizit alle Operationen aus. So sind im vorherigen Abschnitt Vergleichsfunktionen eingeführt worden, die eine boole'sche Spalte mit Informationen über das Resultat eines Vergleiches zu Relationen hinzufügen. Die Selektion an sich wertet daher keine komplexen Bedingungen aus, sondern fokussiert sich auf die Auswertung boole'scher Werte des Prädikates p. Alle Tupel mit Wert *false* in p werden abgestoßen. Damit unterscheidet sich σ_{p} von der Selektion der typischen relationalen Algebra.

2.5 Projektion $\pi_{\mathsf{a}_1:\mathsf{b}_1,\ldots,\mathsf{a}_n:\mathsf{b}_n}$

Führt man nun den Gedanken aus dem vorherigen Abschnitt zu Ende und möchte die für die Selektion generierte *Arbeitsspalte* mit boole'schen Werten wieder aus der Relation entfernen, so hilft der Projektionsoperator. Er übernimmt die Spalten $\mathsf{b}_1, \ldots, \mathsf{b}_n$ der Eingaberelation und benennt diese in $\mathsf{a}_1, \ldots, \mathsf{a}_n$. Hierbei unterscheider sich π von einer gewöhnlichen relationalen Algebra, in der gewöhnlich ein eigener Operator für das Umbenennen von Attributen geführt wird.

2.6 Aggregationen

Die relationale Algebra von Pathfinder umfasst die Aggregationsfunktionen *sum*, *count*, *avg*, *max* und *min*.

Dabei kann die Eingaberelation durch ein Partitionsattribut p eingeteilt sein. In diesem Falle wird die Aggregation für jede einzelne Partition durchgeführt.

2.7 Joins

Die Pathfinder Algebra unterscheidet drei Join-Varianten. Für alle Arten gilt, dass sie über zwei Eingaberelationen verfügen, deren Schemata unterschiedlich sind. Die Algebra sichert also zu, dass ein Spaltenname höchstens in einer Relation vorkommt.

Equi-Join $\bowtie_{a=b}$

Equi-Joins basieren auf einer einzigen Gleichheitsbedingung. Dabei werden jeweils eine Spalte beider Eingaberelationen als Joinkriterium herangezogen.

Semi-Join $\ltimes_{a=b}$

Auch die Semi-Join Operation basiert auf einer einzigen Gleichheitsbedingung. Hier wiederum wird einzig für jedes Tupel der linken Eingaberelation ein Vorhandensein in der rechten Relation geprüft. Bei positivem Ergebnis wird das Tupel der linken Relation in die Ergebnisrelation übernommen.

Theta-Join $\bowtie_{a\theta b}$

Theta-Join ist eine Verallgemeinerung von Equi-Join. Es können mehrere Vergleichskriterien greifen. Für jeden einzelnen Vergleich werden auch hier zwei Spalten der Eingaberelationen betrachtet, Theta-Join bietet dazu die Vergleichsformen $=, <=, <, >, >=$ und $<>$.

2.8 Duplikateliminierung δ

Die Duplikateleminierung δ wird in der Pathfinder Algebra explizit aufgerufen. Dadurch wird die Durchführung dieses häufig kostspieligen Prozess auf notwendige Stellen reduziert. δ macht deutlich, dass Abfragen in Pathfinder ähnlich wie SQL von einer *bag*-Semantik ausgehen, während die klassische relationale Algebra Mengen zugrunde legt, die von Natur aus duplikatfrei sind (*set*-Semantik).

2.9 Mengenoperationen $\setminus$, $\cap$, $\dot{\cup}$

Die Mengenoperationen entsprechen der Mengendifferenz $\setminus$, -schnitt $\cap$ und -vereinigung $\dot{\cup}$. Bei $\dot{\cup}$ gilt es zu beachten, dass das Ergebnis nicht zwingend duplikatfrei ist. Die Mengenvereinigung entspricht dem Befehl `UNION ALL` in SQL. Die Schemata beider Eingaberelationen sind bei allen Mengenoperationen identisch.

2.10 Kreuzprodukt $\times$

Das Kreuzprodukt erzeugt alle Kombinationen der Tupel aus linker und rechter Eingaberelation. Die Schemata der beiden Eingaberelationen sind dabei garantiert disjunkt. Ein Spaltenname erscheint also maximal in einer Relation.

2.11 Tupelnummeriung $\rho_{\langle a_1,\ldots,a_n\rangle,b,p}$

$\rho_{\langle a_1,\ldots,a_n\rangle,b,p}$ fügt der Eingaberelation eine Spalte b hinzu. Für diese Spalte werden Aufzählungsnummern startend bei 1 erzeugt. Die Werte in b sind entsprechend den Attributen $\langle a_1, \ldots, a_n\rangle$ sortiert. ρ ermöglicht ferner die Spezifikation eines Partitionsattribut p, welches bewirkt, dass die Aufzählungswerte unabhängig für Tupel der einzelnen Partitionen durchgeführt werden. Es gibt daher exakt so viele Zahlenreihen wie Partitionen vorhanden sind.

Beispiel 2.2. *Dazu betrachten wir als Beispiel $\rho_{b:\langle a_1,a_2\rangle/p}$ auf einer dreispaltigen Eingaberelation.*

(a) Relation v

a_1	a_2	p
1	2	0
2	5	1
1	3	1
2	4	0
2	1	1

(b) $\rho_{b:\langle a_1,a_2\rangle/p}(v)$

a_1	a_2	p	b
1	2	0	1
2	5	1	3
1	3	1	1
2	4	0	2
2	1	1	2

Tabelle 2.3: Anwendung der Tupelnummerierung $\rho_{\langle a_1,a_2\rangle,b,p}$

Wir partitionieren die Relation nach dem Attribut p *und führen für alle Partitionen jeweils unabhängig eine Nummerierung durch. Dabei müssen*

die Sortierkriterien berücksichtigt werden und zuerst a_1 *und anschließend* a_2 *betrachten werden.*

2.12 Typumwandlung $Cast_{type,n}$

Cast erweitert die Eingaberelation um eine neue Spalte, die das Ergebnis der Umwandlung der Eingabespalte n in Datentyp *type* beinhaltet.

2.13 Dokumentensuche $doc - lookup_{\mathsf{p,n}}$

$doc - lookup_{\mathsf{p,n}}$ ersetzt Strings durch Dokumentenreferenzen für jede Partition definiert durch das Attribut p. Die zugrundeliegende Funktion von $doc - lookup_{\mathsf{p,n}}$ ist $fn : doc$ der XQuery-Spezifikation mit deren Hilfe Daten aus einer externen Datei gelesen werden können.

2.14 Dokumenteneinsicht $doc - access_{\mathsf{p,n}}$

$doc - access_{\mathsf{p,n}}$ führt prinzipiell ein Verbund der rechten Eingaberelation mit den aktiven Fragmenten des Operators durch. Die rechte Eingaberelation referenziert die zu lesenden Tupel. Die Ergebnisrelation enthält schließlich die Eingaberelation erweitert um die gelesenen Daten aus dem Dokument.

2.15 XPath-Step $⏍_{\alpha,n,dup}$

XQuery integriert die *XML Path Language XPath*, um Teile eines XML Dokumentes zu referenzieren. Dazu bietet XPath mit den Lokalisierungspfaden geeignete Hilfsmitel, um in XML Dokumenten ausgehend von sogenannten Kontextknoten zu navigieren. Lokalisierungspfade können in einzelne Schritte separiert werden. Diese Schritte werden in der Pathfinder Algebra durch den XPath-Step Operator $⏍_{\alpha,n,dup}$ dargestellt. Dabei stellen die Parameter von $⏍_{\alpha,n,dup}$ die auszuwertende Achse α und einen Knotentest n dar. Für eine Darstellung aller Achsen und dem Konzept des Knotentests sei auf die XPath Spezifikation des W3C verwiesen ([CD99]). Der Parameter *dup* gibt Information darüber, ob die Ergebnismenge von $⏍_{\alpha,n,dup}$ duplikatsfrei sein muss.

$⏍_{\alpha,n}$ erwartet zwei Eingaberelationen, wobei die erste die aktiven Fragmente des Operators sind und die zweite Informationen zu Partitionen und

Kontextknoten beinhaltet. Das Resultat von $\triangle_{\alpha,n}$ stellen ausgehend von den Kontextknoten die Knoten auf der Achse α dar, die den Knotentest n bestehen. Die Ergebnismenge kann entsprechend des Parameters *dup* Duplikate beinhalten oder nicht.

In Operator $\triangle_{\alpha,n}$ ist das Konzept der Lokalisierungsschritte gekapselt. Er stellt eine Erweiterung der typischen relationalen Algebra dar und lässt sich relational übersetzen, was die Übersetzung zu SQL belegt (siehe dazu [May07]). Mit q verfügen wir über ein Werkzeug, $\triangle_{\alpha,n}$ anderweitig mit Seiteneffekten zu realisieren. Im Rahmen dieser Arbeit werden die Übersetzungen von $\triangle_{\alpha,n}$ für die Achsen `child`, `descendant` und `descendant-or-self` vorgestellt. In Abschnitt 5.17 wird erläutert, welche Vorgänge durch den Operator $\triangle_{\alpha,n}$ im kdb+-Back-End gekapselt sind.

2.16 Konstruktoren Λ_t

XQuery bietet die Möglichkeit neue Fragmente zu erstellen. Diese Fragmente beinhalten wie Dokumentrelationen auch gültige XML Bäume, auf die innerhalb der Abfrage dann auch referenziert werden kann. Die Pathfinder Algebra umfasst zur Erstellung kompletter Fragmente verschiedene Konstruktoren Λ_t der Typen t, die die entsprechenden XQuery Operationen explzit darstellen. Im folgenden werden wir die Konstruktoren für Textknoten, Attribute und Elemente betrachten. Im Zuge dessen werden die darüber hinaus benötigten Konstruktoren des Types *content*, *fcns* und *twig* präsentiert. Jede Fragmentkonstruktion besitzt $\Lambda_{\substack{twig \\ \mathsf{p,n}}}$ als Wurzel, während $\Lambda_{\substack{textnode \\ \mathsf{p,n}}}$, $\Lambda_{\substack{attribute \\ \mathsf{p,n,v}}}$ und $\Lambda_{\substack{content \\ \mathsf{part,n,pos}}}$ immer Blätter im Konstruktorenbaum darstellen.

Textnode-Konstruktor $\Lambda_{\substack{textnode \\ \mathsf{p,n}}}$

Dieser Konstruktor erstellt anhand der String-Werte in n seiner Eingaberelation einen XML-Textknoten. Der Erstellvorgang läuft in Übereinstimmung für jeden Partitionswert in p.

Attribut-Konstruktor $\Lambda_{\substack{attribute \\ \mathsf{p,n,v}}}$

Dieser Konstruktor erstellt ähnlich dem Textnode-Konstruktor XML-Attributsknoten entsprechend der Partitionen p seiner Eingaberelation. Hier müssen nun jedoch String-Werte in n für den Attributsnamen sowie Strings in v als Attributswerte gesetzt werden.

Content-Konstruktor $\underset{\mathsf{part,n,pos}}{\Lambda_{content}}$

Der Content-Konstruktor liest entsprechend seiner Eingaberelation komplette Elemente aus den aktiven Fragmenten. Er stellt damit einen eigenen Operator für das Konzept der *Subtree Copy* dar. $\underset{\mathsf{part,n,pos}}{\Lambda_{content}}$ übernimmt jeweils innerhalb jeder Partition aus part entsprechend der Reihenfolge aus pos alle Knoten referenziert durch n. Dabei werden die Knoten und deren vollständiger Teilbaum bezogen und nachfolgenden Konstruktoren zur Verfügung gestellt.

FCNS-Konstruktor Λ_{fcns}

Λ_{fcns} vereint alle Eingabe-Konstruktoren zu einer Liste. Damit können mehrere konstruierte XML Knoten in einem Element zusammengefasst werden. Der Konstruktor des Types *FCNS - First Child Next Sibling* stellt prinzipiell einen Meta-Operator dar, um die relationale Algebra binär zu halten.

Element-Konstruktor $\underset{\mathsf{p,n}}{\Lambda_{element}}$

Der Element-Konstruktor erstellt für jede Partition p ein eigenes Element entsprechend seiner linken Eingaberelation. Die rechte Eingaberelation referenziert Knoten, die alle basierend auf ihrem Partitionsattribut von den neu zu erstellenden Elementen gekapselt werden bzw. als Unterbaum dieser Knoten angehängt werden.

Twig-Konstruktor $\underset{\mathsf{p,n}}{\Lambda_{twig}}$

Als Wurzel aller Konstruktoren fungiert der Twig-Konstruktor, der die erstellten Fragmente der unter ihm liegenden Konstruktoren abschließend behandelt und Referenzen auf die Wurzeln der neuen Fragmente als Ergebnis erstellt. Die Funktionsweise von Λ_{twig} im kdb+-Back-End wird innerhalb der Übersetzung in Abschnitt 5.18.6 beschrieben.

Beispiel 2.3. *Wir wollen nun das einfache Fragment* `<a>42</a>` *konstruieren. Der Konstruktorenbaum ist folgend dargestellt.*

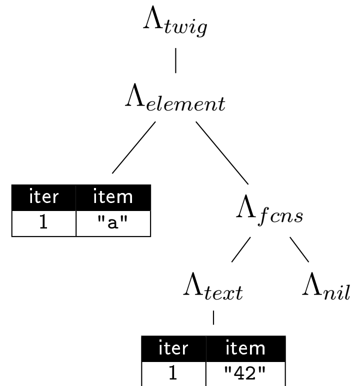


Abbildung 2.1: Konstruktorenbaum

Die Konstruktion des Beispielfragmentes kann bottom-up *betrachtet werden. Beginnend bei der Textknoten-Konstruktion mit Wert* `42` *liefert der folgende Λ_{fcns} das Fragment weiter an $\Lambda_{element}$. Dabei ist zur Illustration im Beispiel Λ_{nil} abgebildet. Dieser Konstruktorentyp liefert ein leeres Fragment, wird in dieser Arbeit jedoch nicht eingeführt, da keine explizite Übersetzung für ihn benötigt wird. $\Lambda_{element}$ hängt den gefundenen Teilbaum mit dem Textknoten* `42` *als Nachfahren des neuen Elementes* `a` *an, während Λ_{twig} das erstellte Fragment abschließend behandelt und Referenzen darauf erstellt.*

Kapitel 3

kdb+/q

kdb+ verfügt über eine eigene Programmiersprache für Datenmanipulation und Datenabfragen. Diese Programmiersprache, q, stellt die Zielsprache der Übersetzung dar. Es ist daher notwendig, die wesentlichen Bestandteile von q, dessen Datenstrukturen und Funktionen zu verstehen. In diesem Kapitel wird zur Beschreibung der Sprachkonzepte von q vorweg die Grundprinzipien des Datenbanksystems kdb+ erläutert und anschließend ein Einblick in Datenstrukturen und Funktionen von q gegeben.

3.1 kdb+

kdb+ verhält sich in vielen Fällen nicht wie ein typisches RDBMS. Die wichtigen Konzepte der *Spaltenorientierung* und der *geordneten Relationen* sind im folgenden erläutert.

Spaltenorientierung

Wichtiges Prinzip von kdb+, das Benutzern typischer RDBMS als signifikanten Unterschied auffält, ist die umgedrehte, vertikale Fragmentierung der Tabellen. Die spaltenorientierte Speicherung der Daten vereinfacht Indizierung, Verbundsoperationen und das Anlegen von Indexen. Daten werden anhand vollständiger, sequentieller Spalten gespeichert, nicht als Zeilen beliebiger Reihenfolge. Dadurch können Aggregationsoperationen und die einfache Selektierung von Spalten enorm beschleunigt werden. Zusätzlich ist jede Spalte unabhängig voneinander was Hinzufügen, Löschen und Abändern einzelner Spalten einfach gestaltet. Darüberhinaus nutzt kdb+ Informationen über den Spalteninhalt wie Sortierung der Daten zur Optimierung von Abfragen.
kdb+ bietet eine Vielzahl an Funktionen, die speziell für komplexe finanzmathematische Berechnungen und Zeitreihenanalysen geeignet sind. Opera-

tionen auf diesen typischerweise zeitlich sortierten Daten profitieren enorm durch die Spaltenorientierung der Tabellen. Dies ist ein Anwendungsbeispiel, bei dem sich das vertikal gedrehte Relationenmodell auszahlt.

Geordnete Relationen in kdb+

Eine Relation stellt Fakten auf einer logischen Ebene dar. Eine Reihenfolge ist daher nur auf logischer Ebene festgelegt, sei es die Schlüsselspalte oder beispielsweise eine Spalte *Name*. Die Definition einer Relation legt jedoch keine Reihenfolge der Tupel fest – auch wenn die Tupel aus physischer Sicht natürlich in irgendeiner Folge auf dem Hintergrundspeicher abgelegt sind (vgl. [EN04, 129]).
In kdb+ im Gegensatz steht q als eine Array-basierte Programmier- und zeitbezogene Anfragesprache zur Verfügung (vgl. [Sys06]). Wie *Array-basierte Programmiersprache* bereits andeutet, können mit q Elemente mit *positional look-ups* aufgesucht werden, seien dies Elemente einer Liste, Zeilen einer Tabelle, oder Elemente von Zeilen einer Tabelle. Es handelt sich bei Listen oder Tabellen in q um geordnete Mengen von Daten.

Im Kontext dieser Arbeit zeichnet sich kdb+ durch das zugrundeliegende Datenmodell und die Erweiterbarkeit seiner Skriptsprache q als ein geeignetes Zielsystem aus für die Übersetzung der Operatoren wie in Kapitel 2 vorgestellt. Im nächsten Abschnitt werden die grundlegenden Datenstrukturen wie Liste und Tabelle sowie die Operatoren von q. Dabei wird der Fokus auf die Operationen gelegt, die zur Übersetzung der Algebra von Belang sind und dazu benutzt werden. Für weitere Aspekte von q sei auf die Referenzen von Kx Systems [Sha05], [Ort06] und deren Partner [O'N06] verwiesen.

3.2 q

kdb+ wird mit einer eigenen Skriptsprache q für Datenbankabfragen und Programmierung benutzerdefinierter Funktionen ausgeliefert.
q umfasst atomare Datentypen wie Ganzzahlen, Fließkommazahlen, Strings und Symbole.

Zuweisung in q geschieht durch einen Doppelpunkt `(:)` [1]. Variablendeklaration und Wertzuweisung geschehen in einem einzigen Schritt. Die q

[1] *Hinweis:* Die Klammerung soll lediglich q Operatoren, die einzig aus Sonderzeichen bestehen, vom Text abgrenzen und gehört nicht zur Syntax von q.

Eingabeaufforderung ist dargestellt durch `q)` und in den folgenden Beispielen führend in jeder Zeile, in der Benutzereingaben getätigt werden. Es gilt dabei zu beachten, dass spezifizierte Werte einer Zuweisung und Ausdrücke mit endendem Semikolon nicht auf der Konsole ausgegeben werden. Wir betrachten nun ein Beispiel zur Zuweisung der Ganzzahl `3` an die Variable `a`. Anschließend wird der Wert der Variablen `a` abgefragt.

Listing 3.1: Zuweisung in q

```
1 q) a:3;
2 q) a
3  3
```

Ausführungsreihenfolge in Q ist grundsätzlich von rechts nach links. Durch Klammerung können Ausdrücke strukturiert werden.

Listen sind geordnete Ansammlungen atomarer oder anderer Datentypen einschließlich Listen. Sie werden dargestellt durch in Klammern mit Semikola getrennte Elementen. Anhand des Code-Beispiels in Listing 3.2 sehen wir in Zeile 1 wie in q eine Liste mit den Elementen `1`, `2` und `3` erstellt und der Variablen `a` zugewiesen wird. Das Erstellen von einelementigen Listen ist in q nur mit der Funktion `enlist` möglich. Die zweite Zeile des Code-Ausschnittes demonstriert dies. Dort wird eine Liste mit nur einem Element, `‘einElement` erstellt.

Listing 3.2: Listen in q

```
1 q) a:(1;2;3);
2 q) b:enlist ‘einElement;
```

`‘einElement` ist ein Element des Datentypes Symbol. Elemente dieses Types sind durch das führende `(‘)` gekennzeichnet und stellen das Äquivalent zu Strings in anderen Programmiersprachen dar. Dabei gilt strikt zwischen den q Datentypen Symbol und String zu unterscheiden. Wir *sparen* uns eine Dimension bei der Benutzung von Symbolen im Vergleich zu Strings, da String lediglich ein anderer Begriff für eine Liste von Elementen des Types *Char* ist. Im Gegensatz dazu ist Symbol ein atomaren Datentyp, der nur als ganzes anzusehen ist und für den in kdb+ zudem spezielle Ablagemechanismen integriert sind. Um Strings zu Symbols zu transformieren, wird die *Cast*-Funktion in q `($)` mit der Symbol-Identifikation ‘ als ersten Parameter sowie dem Eingabe-String als zweiten. Mit `‘$"myString"` wir die Zeichenkette `"myString"` zu einem *Symbol* umgewandelt. Die entgegengesetzte Richtung funktioniert mit der einstelligen Funktion `string`, welche darüber hinaus auf alle Datentypen von q anwendbar ist und die String-Darstellung des Wertes

zurückgibt.
In Zeile 3 sind zwei Funktionen zu sehen: Die Konkatenation `(,)` und `count`.

```
3 q) count a,b
4  4
```

Hierbei gilt es die Ausführungsreihenfolge von rechts nach links zu beachten. Es werden zuerst die Listen `a` und `b` konkateniert bevor die Länge der entstandenen Liste gemessen wird mit `count`. Indizierung von Elementen geschieht im folgenden in Zeile 5. Es können auch mehrere Elemente auf einmal angesprochen werden. Dazu muss eine Liste als Index angegeben werden wie in Zeile 7.

```
5 q) a[1]
6  2
7 q) a[(0;2)]
8 1 3
```

Zeile 9 zeigt schließlich die Anwendung der primitiven Addition, die wie zu erwarten zu jedem Element der Liste `3` addiert und als Ergebnis zurück gibt.

```
9  q) a + 3
10  4 5 6
11 q) key 3
12  0 1 2
```

Listen lassen sich darüber hinaus auch mit der *take*-Funktion `#` erstellen. x#y wandelt y in eine Liste der Länge x um. Des weiteren, generiert `key` eine Zahlenreihe von 0 bis zu dem Wert des Parameters.

In q stehen Adverbien zur Verfügung, die die Wirkungsweise von Funktionen modifizieren. Diese Adverbien sind:

`each` *einstelliges each* Für einstellige Funktionen wird die Funktion durch `each` auf jedem Element des Parameters ausgeführt.

`(/:)` *each-right* Führt die Funktion für das gesamte erste Argument auf jedem Element des zweiten Argumentes aus.

`(\:)` *each-left* Umgekehrt zu `(/:)`

`(')` *each-both* Führt die modifizierte Funktion elementweise auf korrespondierenden Elementen aus.

Listen können auch mehr als eine Dimension besitzen und wiederum Listen als Elemente führen. Das Indizieren dieser Elemente erfolgt mit

`[(`$x_1, \dots, x_n$`);(`$y_1, \dots, y_n$`);...]`, wobei $x_1, \dots, x_n$ Elemente der ersten Ebene referenzieren, $y_1, \dots, y_n$ der zweiten, etc. In Listing 3.3 sehen wir diese Art der Indizierung. In der ersten Zeile wird der Variablen `a` eine zweielementige Liste zugewiesen, deren Elemente wiederum Listen der Länge drei bzw. zwei sind. Zeile 2 zeigt die Indizierung des zweiten Wertes der ersten Liste von `b`[2].

Listing 3.3: Mehrdimensionale Listen in q

```
1 q) b:((1;2;3);(4;5));
2 q) b[0;1]
3  2
```

Ein weiteres Beispiel der Indizierung wird in Zeile 4 gezeigt. Dabei verwenden wir wieder die Liste `a` mit den Werten erneut `(1;2;3)`. Eine mehrdimensionale Liste wird nun zur Indizierung von Elementen der obersten Ebene verwendet. Die Indexliste besitzt die drei Elemente `0`, `2` und die Liste `(0;1)`. Die Ergebnisliste erhält die Gestalt der Indexliste.

```
4 q)a[(0;2;(0;1))]
5 1
6 3
7 1 2
```

Zeile 4 stellt die Anwendung von `each` auf dem Operator `count` dar. Dieser gibt anstatt der Länge Liste auf oberster Ebene die Länge jeder einzelner Listen zurück.

```
8 q) count each b
9  3 2
```

Die Anwendung des Adverbs *each-right* wird an der Konkatenation `(,)` gezeigt. Das Adverb bewirkt in Zeile 10, dass jedes Element des zweiten Argumentes an die Liste im ersten Argument angehängt wird. Das Ergebnis entspricht der Kardinalität des zweiten Argumentes.

```
10 q)(1;2) ,/: (3;4)
11 1 2 3
12 1 2 4
```

Der Operator `raze` klappt die zwei höchsten Level zu einer Dimension zusammen.

```
13 q) raze b
14  1 2 3 4 5
```

[2] Alternativ kann *Indizierung in die Tiefe* durch die Punktnotation `(.)` erreicht werden. So liefert `b . (0;1)` das gleiche Ergebnis wie `b[0;1]`. Dies ist hilfreich, wenn mehrere Indizierungen auf einmal mit *each-right* durchgeführt werden sollen `(./:)`

Wörterbuch

Das Konzept eines Wörterbuches - *Dictionary* - ist im einfachsten Fall eine Benennung der Elemente einer Liste.

Listing 3.4: Wörterbücher in q

```
1 q)`a`b`c!(4;5;6)
2 a| 4
3 b| 5
4 c| 6
```

Wörterbücher sind im wesentlichen Listen mit dem Unterschied, dass der Domänenbereich eine beliebige Menge von unterschiedlichen Elementen anstatt Indizes darstellt. Die Erstellung erfolgt mit dem Operator `(!)`, welchem die Domänenliste vorangestellt und die Werteliste folgt, wie im Beispiel `` `a`b`c!(4;5;6) ``[3]. Um auf den Domänenbereich zuzugreifen, steht die Funktion `key` zur Verfügung, der Wertebereich lässt sich mit `value` ansprechen.

Gruppierungsoperator `group`

Der Gruppierungsoperator `group` gruppiert die Elemente seines Argumentes und erstellt ein Wörterbuch, das als Domäne die eindeutigen Elemente umfasst und dessen Wertebereich die Indizes sind, an denen die Elemente in der Eingabeliste auftauchen. Das Listing aus 3.5 führt `group` auf einem String, also einer Liste von atomaren Werten des Types *Char*, aus. `"misp"` ist hierbei der aus vier Elementen bestehende Domänenbereich. Hier gilt noch auf die besondere Art der Ausgabe `,0` hingewiesen, womit q ausdrückt, dass es sich um eine einelementige Liste mit dem einzigen Element `0` handelt.

Listing 3.5: Anwendung der `group`-Funktion

```
1 q)group "mississippi"
2  "misp"!(,0;1 4 7 10;2 3 5 6;8 9)
```

Spaltenwörterbücher

Eine sehr nützliche Art von Wörterbüchern sind Spaltenwörterbücher oder *column dictionaries*. Diese bilden eine Liste von Symbolen auf eine Liste von Listen gleicher Kardinalität ab. Diese Wörterbücher besitzen folglich die Form $s_1 \ldots s_n!(v_1; \ldots; v_n)$, wobei s_i ein Symbol ist und alle v_i Listen gleicher Länge sind. Somit wird ein Bezug zwischen einem bestimmten Symbol und einer Werteliste hergestellt, was uns als Grundlage für Tabellen in q dient.

[3] Die Schreibweise `` `a`b`c `` ist hier korrekt. Eine Liste von Symbolen kann mit direkt aufeinander folgenden Elemente geschrieben werden.

Tabelle

Tabellen sind transponierte *column dictionaries*. Durch die Funktion `flip` wird die Transposition durchgeführt. Dabei wird keine tatsächliche Neuordnung der Daten im Hintergrund von kdb+ vollzogen, sondern einzig die Reihenfolge der Indizes invertiert. Diese Tatsache macht deutlich, dass Tabellen in kdb+ *column-oriented* sind und die Daten spaltenweise abgelegt sind.

Tabellen können auch ohne den Umweg über *column dictionaries* erstellt werden. Mit `([]` c_1`:`L_1`;...;`c_n`:`L_n`)` wird eine Tabelle mit den Spaltennamen c_i und der korrespondierenden Liste L_i mit den Spaltenwerten erstellt. Alle L_i besitzen hierbei wieder gleiche Länge. Die eckigen Klammern umfassen den Primärschlüssel der Tabelle, bei dieser Schreibwese ist kein Schlüssel für die Tabelle definiert.
Eine Tabelle ist eine der Reihe nach sortierte Folge von Datensätzen. Ein Datensatz besteht hierbei aus einer Abbildung der Spaltennamen auf die Werte einer Zeile.

Tabellenabfragen

q besitzt eigene Funktionen zur Manipulierung von Tabellen. Die grundlegende Syntax von Abfragen lautet

- `select` p_s [`by` p_g] `from` p_t [`where` p_w] Wählt für die Tabelle p_t die Tupel, für die Bedingung p_w wahr ist, gruppiert diese anhand von p_g und selektiert die Spalten ausgedrückt in p_s. Soll die Ergebnistabelle alle Spalten p_t umfassen, so kann p_s weggelassen werden. `by` und `where` sind optionale Bestandteile.

- `exec` p_s [`by` p_g] `from` p_t [`where` p_w] Wie `select`, liefert jedoch die Ergebniswerte in Listenform, wenn p_s eine Spalte darstellt, ansonsten als *Dictionary*.

- `update` p_s `from` p_t `where` p_w Modifiziert bestehende Spalten und fügt noch nicht vorhandene zur Tabelle hinzu. `where` ist optional.

- `delete` p_s `from` p_t `where` p_w Bei `delete` darf nur entweder p_s oder p_w definiert sein. Für den ersten Fall wird die Tabelle ohne die definierten Spalten zurückgegeben, ansonsten die Tabelle ohne die Zeilen für die die `where`-Klausel wahr sind.

Das Muster zur Erstellung einer `select`-Abfrage wird in Listing 3.6 angewendet. Diese Abfrage gibt die Summe aller Werte `qty` für jede `id` zurück,

deren entsprechende Wert in `rate` mindestens `0.3` ist und `code` mit dem Zeichen `A` beginnt. Es gilt zu beachten, dass Ausdrücke innerhalb der `select` und `where` Klauseln entgegengesetzt der erwähnten, allgemeinen Evaluierungsreihenfolge von links nach rechts abgearbeitet werden. Die Operation `0!` wird jedoch wie erwartet auf das Resultat der `select`-Abfrage angewandt. Es entfernt explizite Schlüsselbeziehungen einer Tabelle. Im Beispiel liefert die `select`-Abfrage eine Tabelle mit Schlüsselspalte `id`, auf der die Gruppierung stattfand. Dieser Schlüssel wird dann durch `0!` wieder eliminiert.
Die links nach rechts Evaluierung in der `where`-Klausel hat zur Folge, dass zuerst der erste Filter ausgewertet und auf der resultierenden Tabelle die weiteren Bedingungen getestet werden. Es empfiehlt sich daher den Filter mit der höchsten Selektivität als erste Bedingung in der `where`-Klausel zu führen.

Listing 3.6: Select-Abfrage

```
1  q)t
2  id  qty   price rate code
3  -------------------------
4  101 100   10.3  0.3  A
5  105 20000 0.8   0.5  AB
6  101 5000  24.9  0.4  AC
7  107 1000  0.85  0.29 BC
8  q)0!select qtySum:sum qty by id
9      from t
10     where (rate>=0.3), code like "A*"
11 id  qtySum
12 ----------
13 101 5100
14 105 20000
15 q)exec qty from `id xasc t
16 100 5000 20000 1000
```

Um auf einer bestehenden Tabelle `t` oder dem Ergebnis einer Abfrage eine Sortierung durchzuführen, steht in q die Funktion `xasc` zur Verfügung. Ihr erster Paramter sind dabei die Sortierspalten. Zeile 15 aus Listing 3.6 zeigt wie `t` zuerst nach Spalte `id` sortiert wird. Anschließend wird `qty` selektiert und durch `exec` in Listenform ausgegeben.

Verbundoperationen

q bietet Equi-Joins, Left-Joins, und einige weitere Verbundoperationen. Besteht eine Fremdschlüsselbeziehung zwischen zwei Tabellen, so lässt sich ein Verbund besonders einfach herstellen.
Für die Operatorenübersetzung benötigen wir den Operator `lj`, der prinzipiell einen *Left Outer Join* darstellt. Hierbei gilt es jedoch zu beachten, dass q für `lj` erwartet, dass die Verbundsspalte der zweiten Relation eine Schlüsselspalte darstellt und damit nur eindeutige Werte besitzt.

Attribute

In q können wir Listen und damit auch Spalten von Tabellen mit den Attributen `‘s#`, `‘u#` und `‘g#` versehen. Diese Attribute sind gedacht, um Speicheranforderungen zu reduzieren und die Elementauffindung zu beschleunigen. Der q Interpreter ist in der Lage bestimmte Optimierungen anzuwenden, sobald er mit Attributen versehenen Daten stößt. Für Listen mit eindeutigen Werten empfiehlt sich `‘u#`, für sortierte Listen das `‘s#`-Attribut und schließlich `‘g#` für Listen mit sich häufig wiederholenden Werten. `‘s#` bewirkt den Wechsel von gewöhnlicher, linearer Suche zu binärer Suche von Elementen. `‘u#` beschleunigt Operationen wie `distinct` und bewirkt vorzeitiges Abschließen von Vergleichsoperationen. Das Gruppierungsattribut `‘g#` veranlässt q zur Erstellung eines Indexes auf der Liste. Der Index ist ein Wörterbuch, das zu allen eindeutigen Werten die Indizes der Vorkommen in der Liste speichert (vgl. `group`-Operator in Abschnitt 3.2).

Benutzerdefinierte Funktionen

Im Zuge der Algebra-Übersetzung werden auch eigens definierte Funktionen eingeführt. Eine Funktion `f` mit dem Parametern `x`, `y` und `z`, die die Summe der drei Parameter berechnet ist in Listing 3.7 dargestellt. Der Rückgabewert wird in Funktionen durch den führenden Doppelpunkt in der letzten Zeile gekennzeichnet. Ein beispielhafter Aufruf lautet `f[2;3;4]`.

Listing 3.7: Funktionsdefinition

```
1 f:{[x;y;z];
2  : x + y + z;
3  };
```

Die Funktion hat ohne explizite Parameterangabe – `f:{ : x + y + z; };` – die gleiche Funktion. Sind keine Paramter angegeben, so geht q von drei impliziten Funktionsparametern `x`, `y` und `z` aus.

Kapitel 4

XML Daten in kdb+

Das zugrundeliegende Paradigma von XML Daten sind Bäume. Logisch gesehen sind XML Daten daher hierarchisch aufgebaut und können als Bäume repäsentiert werden. Knoten in diesen Baumrepräsentationen entsprechen Elementen im XML Dokument. Auch Attribute dieser Elemente, Textinhalte, Kommentare und Prozessoranweisungen sind als Baumknoten erfasst. Anhand dieser Vorgehensweise lässt sich jedes XML Dokument äquivalent als Baum darstellen. Entsprechend der Baumrepräsentationen lassen sich XML Dokumente schließlich relational darstellen (siehe dazu [Gru02]). In folgenden Abschnitten wird die Repräsentation von XML-Dokumente in kdb+ dargestellt. Dies wird zuerst anhand eines einzelnen Fragmentes dargestellt, bevor die Handhabung einer Vielzahl von Fragmenten und ganzen XML-Dokumenten vogestellt wird.

4.1 Relationales Layout

Wir benutzen eine ähnliche Kodierung von XML Fragmenten wie sie in [GST04] vorgeschlagen wurde. Dabei besteht ein Baumknoten v aus folgenden Informationen:

pre Eindeutige Pre-Order Traversierungsnummer von v innerhalb des gesamten Baumes (siehe dazu auch [Gru02])

size Anzahl der Knoten im Teilbaum unterhalb von v

level Länge des Pfades von der Wurzel bis zu v

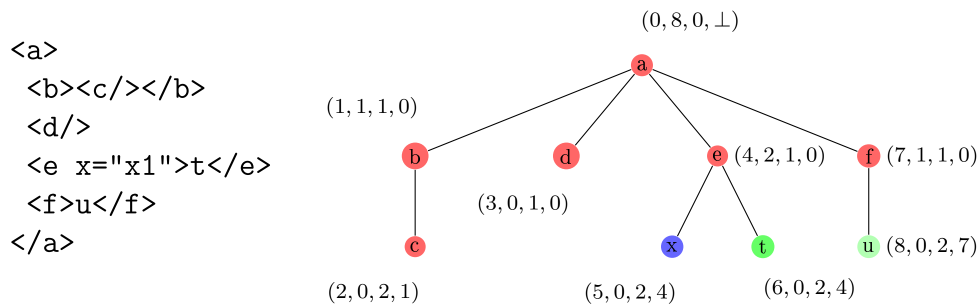
parent Referenz auf w.pre des Elternknotens w von v

kind Knotentyp

name *Tag*-Name eines Elements oder Name eines Attributes

val Wert eines Text oder Attributknotens

Beispiel 4.1. *Wir wollen nun die Baumrepräsentation anhand des XML-Fragmentes in Abbildung 4.1(a) zeigen. Abbildung 4.1(b) zeigt alle Knoten des XML-Fragmentes mit* (pre,size,level,parent)*-Kodierungsinformationen. Dabei sind Elementknoten rot, Textknoten grün und Attributsknoten blau dargestellt.*



(a) XML-Fragment

(b) Baumrepräsentation von (a)

pre	size	level	parent	kind	name	val
0	8	0		1	‘a	
1	1	1	0	1	‘b	
2	0	2	1	1	‘c	
3	0	1	0	1	‘d	
4	2	1	0	1	‘e	
5	0	2	4	2	‘x	‘x1
6	0	2	4	3		‘t
7	1	1	0	1	‘f	
8	0	2	7	3		‘u

(c) Relationale Kodierung von (a)

Abbildung 4.1(c) zeigt schließlich die Darstellung als Relation. Auf diese Art und Weise werden XML Fragmente als Tabellen im kdb+-Back-End angelegt.

Anhand von pre, size, level besitzt man ausreichend Informationen, um die strukturelle Information der XPath-Achsen abzubilden. Für einen Ausschnitt der Achsen ist dies in Tabelle 4.1 illustriert. Dabei wird ausgehend von einem Kontextknoten v beschrieben, welche Bedingungen erfüllt sein müssen, damit der Knoten res auf der jeweiligen Achse α liegt. Es ist ersichtlich, dass für

die Achse `child` einige Bedingungen zu erfüllen sind. Im Zuge der Übersetzung der Algebra wird bemerkbar, dass diese Umsetzung auf kdb+/q effizient nicht unbedingt trivial zu bewerkstelligen ist. Dies ist der Grund für die Hinzunahme der parent-Werte, mit welchen eine alternative Definition dieser Achse — dargestellt als `child (2)` in Tabelle 4.1 — betrachtet werden kann.

Achse α	Achsenprädikate
`descendant`	$v.\mathsf{pre} < res.\mathsf{pre} \leq v.\mathsf{pre} + v.\mathsf{size}$
`descendant-or-self`	$v.\mathsf{pre} \leq res.\mathsf{pre} \leq v.\mathsf{pre} + v.\mathsf{size}$
`child` (1)	$v.\mathsf{pre} < res.\mathsf{pre} \leq v.\mathsf{pre} + v.\mathsf{size}$
	$\wedge\ res.\mathsf{level} = v.\mathsf{level} + 1$
`child` (2)	$res.\mathsf{parent} = v.\mathsf{pre}$

Tabelle 4.1: XPath-Achsenprädikate

Zur Durchführung von Knotentests der XPath-Steps werden die Werte der Knotenarten in kind sowie der Knotenwerte in name zu Rate gezogen.

Die im folgenden Kapitel 5 beschriebenen Übersetzungen basieren auf diesem Datenlayout für XML Fragmente. In Kapitel 6 werden Varianten für die Evaluierung der Achse `child` gegenüber gestellt, um die Existenz der Spalte parent und die damit involvierte Verwaltung dieser Zusatzinformationen zu rechtfertigen.

Darstellung ganzer Dokumente

Dokumente besitzen Bezeichnungen, die für gewöhnlich den ursprünglichen Dateinamen der XML-Dokumente entsprechen. Für diese Bezeichnung besitzt jedes XML-Dokument in seiner relationalen Darstellung einen Wurzelknoten vom Typ *Dokumentknoten* mit der Bezeichnung des Dokumentes als Wert der Spalte val. An diesen Knoten wird das XML-Fragment wie in Abschnitt 4.1 als Unterteilbaum angehängt.

Attribute der Dokumentenspalten

Es lässt sich bereits jetzt vorhersagen, dass die pre-Spalte eine elementare Rolle besitzt. Sie kann als Primärschlüssel der Knoten gesehen werden und sämtliche Referenzen innerhalb der XQueries werden auf den Pre-Wert eines Knoten zeigen. Wir nutzen an dieser Stelle aus, dass Tabellen in kdb+ *order-aware* sind, also dass Einträge einer Tabelle einzig anhand ihrer Position in

der Tabelle angesprochen werden können (*positional lookups*).
Geht man nun davon aus, dass die pre-Werte eines XML Fragmentes im Intervall $[0, n-1]$ liegen, wobei n die Anzahl der Knoten im Fragment darstellt, so kann man durch Sortierung der Fragmenttabelle bezüglich der pre-Spalte erreichen, dass alle pre-Werte identisch auf die Zeilennummern aus $[0, n-1]$ des Tupels innerhalb des Fragmentes abgebildet werden. Wählt man nun ein Tupel mit bestimmten pre-Wert, so verzichtet man komplett auf wertebezogenes Nachschlagen des Tupels. Dies bringt bei Operationen, die Zugriffe auf die Fragmente der Abfrage vollziehen, enorme Vorteile. Für die Operatoren $doc - lookup$ und $\triangle_{\alpha,n}$ benötigten Werte aus den Fragmenten können die pre-Werte der relevanten Tupel als Indizes auf den Fragmenttabellen bzw. der relevanten Spalten dieser Tabellen benutzt werden.
Die Spalten parent und name werden zusätzlich noch mit dem *Grouping*-Attribut (`g#) ausgestattet. Beide Spalten werden für wertbezogenes Nachschlagen verwendet, was durch die vorhergehende Attributierung mit `g# beschleunigt wird.

4.2 Mehrfache Fragmente

Der Gültigkeitsbereich einer XQuery-Abfrage kann sich über mehrere XML Fragmente spannen. Einerseits können dies Dokumentenfragmente sein, andererseits können durch die in Abschnitt 2.16 eingeführten Konstruktoren auch neue Fragmente dynamisch erzeugt werden. Daher müssen wir einen Weg finden, Dokumentenrelationen, aktive Fragmente und erzeugte Fragmente handhaben, unterscheiden und referenzieren zu können.

Fragmentliste $\mathcal{F}$

Es bestehen mehrere Möglichkeiten zum Umgang mit mehreren Fragmenten. Einerseits können wir wie in [GST04] beschrieben, aktive (geladene bzw. erzeugte) Fragmente in einer einzigen Relation halten, die zusätzlich zu der relationalen XML Kodierung wie im vorherigen Abschnitt beschrieben noch eine fragment-Spalte besitzt. Die Zugehörigkeit eines Knotens zu einem bestimmten Fragment wird über diese Spalte dann angegeben. Diese Vorgehensweise würde durchaus mit dem Datenmodell von kdb+ übereinstimmen, dass besonders effizient auf Tabellen operieren kann. Jedoch bedeutete dies, dass wir nicht mehr per *positional lookups* Knoten referenzieren könnten, da pre-Werte dann von ihrem Index in der Tabelle abweichen könnten. Eine ähnlich schwerwiegende Folge entstünde aus der Tatsache, dass durch Anfügen von Tupeln an bestehende Tabellen dies zum Verlust der Attribute der Spal-

ten führt. Die Spalten pre,parent und name würden dadurch ihr Sortierungs- bzw. Gruppierungs-Attribut verlieren. Folgende Operationen auf der originalen Dokumentenrelation würden unter beträchtlichen Geschwindigkeitseinbußen leiden.
Um dies zu vermeiden, beschreiten wir im kdb+-Back-End den Weg der Verwaltung einer globalen Fragmenteinheit $\mathcal{F}$. $\mathcal{F}$ wird sich grundsätzlich als eine *Liste von Fragmenten* herausstellen. Diese Liste enthält eine Tabelle für jedes Fragment. Dadurch ist jedes Fragment eine in sich abgeschlossene Einheit in Form einer Tabelle nach dem Schema aus Abschnitt 4.1. Dokumentenrelationen behalten ihre Attribute, da sie im Gültigkeitsbereich einer Anfrage unverändert bestehen bleiben. Zusätzlich entsprechen alle pre-Werte grundsätzlich den Zeilenindizes der Tupel in der Tabelle. Die Identifikation und Referenzierung der Fragmente sind eindeutig bestimmt durch die Position – dem Index – des Fragmentes in $\mathcal{F}$.
Pathfinder verfügt zur Fragmentverwaltung über drei Operatoren *FRAG*, *ROOTS* und *FRAG_UNION*. Die Implementierung der Fragmenthandhabung mit Seiteneffekten in kdb+ ermöglicht, dass alle drei Operatoren im Zuge der Übersetzung außen vor bleiben können, weshalb auch nicht weiter auf sie eingegangen wird.

Referenzierung von Knoten aus mehreren Fragmente

kdb+ ist in der Lage, nicht-atomare Elemente als Bestandteile von Listen oder Tabellen zu führen. Eine Knotenreferenz lässt sich daher als Tupel (frag, pre) darstellen, wobei jeder Knoten eines Fragmentes eindeutig beschrieben wird. Die Fragmentnummer entspricht der Stelle des Fragmentes in der globalen Fragmenteneinheit $\mathcal{F}$, die pre-Nummer bezieht sich dann auf den Index des Knotens in dieser Fragmenttabelle.

Dokumenteneinheit $\mathcal{D}$

Komplette XML-Dokumente, die persistent in der Datenbank gehalten werden, sind im kdb+-Back-End in einem eigenen q Namensbereich angesiedelt, um isoliert von einer Abfrage bestehen zu können. Der Zugriff auf diesen Namensbereich soll einzig durch eine globale Dokumenteneinheit $\mathcal{D}$ geschehen. Diese Dokumenteneinheit repräsentiert eine Liste von Tabellen, wobei jede Tabelle ein persistent gespeichertes Dokument der Datenbank darstellt.
$\mathcal{D}$ lässt sich als eine Funktion `docs` in q ausdrücken. Dabei geht `docs` davon aus, dass der q Namensbereich für alle persistenten Dokumente `` `.pf.doc `` entspricht.

Listing 4.1: Repräsentation von $\mathcal{D}$ in q

```
docs:{[];
 shortIDs:tables `.pf.doc;
 longIDs:`$ string[`.pf.doc.] ,/: string[shortIDs];
 :value each longIDs;
 };
```

Listing 4.1 beschreibt wie die Funktion `docs` mit der eingebauten q Funktion `tables` für den Namensbereich `` `.pf.doc `` die q Bezeichnungen der dort verfügbaren Tabellen abfragt und in `shortIDs` ablegt. Um den vollständigen Namensbezeichner der Tabellen zu erlangen, werden an den Namensbereich alle diese `shortIDs` angehängt. Durch `value each` werden die durch `longIDs` referenzierten Tabellen als Liste zurückgegeben. Damit erhalten wir exakt die gewünschte Datenstruktur für $\mathcal{D}$.

Kapitel 5

Operatoren-Übersetzung

Die relationale Algebra wie in Kapitel 2 vorgestellt impliziert einen Abarbeitungsplan. Die Operatoren dieses Planes wurden dazu vorgestellt. Für die Ausführung einer XQuery auf dem relationalen Back-End kdb+ wird dieser Plan nun abgearbeitet. Dadurch muss für jeden Operator eine Abbildung auf kdb+ gefunden werden. In diesem Kapitel wird die Übersetzung der Operatoren der relationalen Algebra auf reell ausführbaren q Code beschrieben. Dazu wird nach einer Erläuterung des Abarbeitsplanes noch benötigte Hilfsmittel zur Definierung des Übersetzungsvorganges beschrieben. Damit sind schließlich alle Voraussetzungen gegeben, um die Operatoren detailliert zu übersetzen.

5.1 Abarbeitungsplan

Der Anfrageplan, der durch die relationale Algebra von Pathfinder impliziert ist, entspricht von der Struktur her einem azyklischen Digraphen oder *DAG*. Ein Digraph ist ein gerichteter Graph, der azyklisch heißt, wenn er keinen gerichteten Kreis enthält (vgl. [Ste01]). Zur Übersetzung können wir einen *DAG top-down* traversieren, während die Zusammensetzung der Ergebnisse der Operatoren *bottom-up* geschieht. Die DAG-Struktur hat zur Folge, dass sich mehrere Operatoren die gleichen Kinder teilen. So können Ergebnisse einer Operation mehreren anderen Operatoren zur Verfügung gestellt werden. Aufgrund der binären Algebra in Pathfinder besitzen Operatoren immer maximal zwei Kind-Operatoren. Der Startpunkt für der Traversierung des Anfrageplans wird durch den Operator *Serialisierung* dargestellt. *Serialisierung* liefert das Ergebnis einer Anfrage sortiert nach einem Reihenfolgenattribut und rekonstruiert entsprechend aus der Ergebnismenge und den Fragmenten der Abfrage ein gültiges XML-Fragment. Auf *Serialisierung* wird im Rahmen

dieser Arbeit nicht weiter eingegangen.

5.2 Folgerungsregel

Die Übersetzung der Operatoren in diesem Kapitel wird anhand von Folgerungsregeln beschrieben. Diese Regeln besitzen dabei folgende Form:

$$\mathcal{F} \vdash op \Mapsto (\mathcal{E}, \mathcal{Q}, \mathcal{F}_1) \tag{5.1}$$

Sie besagen, dass bei gegebenem $\mathcal{F}$ als globale Fragmentliste der Abfrage die Anwendung des Operators *op* übersetzt wird in

- eine Variablenumgebung $\mathcal{E}$,
- den erzeugten q Code $\mathcal{Q}$ und
- eine möglicherweise modifizierte globale Fragmentliste $\mathcal{F}_1$.

$\mathcal{E}$ stellt eine Abbildung von Spaltennamen auf q Variablen dar. Eine Abbildung in $\mathcal{E}$ wird nach dem Schema $\mathsf{var}_{\mathsf{algebra}} \rightarrow \mathrm{var}_{\mathrm{q}}$ dargestellt.

Operationen auf $\mathcal{F}$ bestehen lediglich aus einem Anhängen neuer Fragmente an das Ende von $\mathcal{F}$, was dargestellt wird als $(\mathcal{F}; \{frag_{neu}\})$.
Eine Vereinigung von bestehendem und neuem q-Code wird durch die Anwendung des Operators $\|$ bewerkstelligt. $\mathcal{Q}_v \| \mathcal{Q}_{op}$ fügt $\mathcal{Q}_{op}$ an vorliegenden Code $\mathcal{Q}_v$ eines vorhergehend bereits vollständig übersetzten Knotens v.
Jeder Abarbeitungsplan der Pathfinder Algebra beginnt mit $\mathcal{F} = \emptyset$.
Darüber hinaus ist die dauerhafte Dokumenteinheit $\mathcal{D}$ aus Abschnitt 4.2 Bestandteil des Geltungsbereiches einer Abfrage. Sie repräsentiert die persistent abgelegten Dokumente innerhalb der Datenbank.

5.2.1 Y-Operator

Da der Abarbeitungsplan einer Anfrage eine DAG-Struktur besitzt, können Probleme entstehen bei Operatoren mit mehreren Eingängen. Das rekursive Aufrufen von Teilen des *DAG* während der Operatorenübersetzung kann zur Mehrfachausführung der Übersetzung eines Operatoren führen. Um dies zu vermeiden fügen wir über alle Operatoren einen Y-Operator hinzu, der die Koordination der Übersetzung des ihm zugeordneten Operators übernimmt.

$Y_{id,env}$ führt die Parameter *id* als eindeutige Identifikation eines bestimmten Y-Operators und eine Referenz auf eine globale Nachschlagetabelle *env*. *env* bildet die Identifikationsnummern der Y-Operatoren auf die Variablenumgebung $\mathcal{E}$ des unter ihm liegenden Operators der relationalen Algebra ab,

soweit dieser bereits evaluiert wurde. Folgende Regel wird verwendet, wenn $Y_{i,env}$ noch nicht ausgeführt wurde und daher (if $env(id)$ is $\emptyset$) gilt — also das Nachschlagen des Wertes für id in env kein Ergebnis liefert.

$$\frac{\begin{array}{c}(\text{if } env(id) \text{ is } \emptyset)\\ \mathcal{F} \vdash v \Rrightarrow (\mathcal{E}, \mathcal{Q}, \mathcal{F}_1)\\ env(id) \lhd \mathcal{E}\end{array}}{\mathcal{F} \vdash Y_{id,env}(v) \Rrightarrow (\mathcal{E}, \mathcal{Q}, \mathcal{F}_1)} \tag{5.2}$$

Sollte $Y_{id,env}$ noch nicht in env gelistet sein, so wird der Kindoperator v ausgeführt und $(\mathcal{E}, \mathcal{Q}, \mathcal{F}_1)$ weitergegeben. Des weiteren wird, beschrieben durch $env(id) \lhd \mathcal{E}$, unter dem Schlüssel id die Variablenumgebung $\mathcal{E}$ in env eingefügt. Sollte $Y_{id,env}$ bereits ausgeführt worden sein, tritt Regel 5.3 in Kraft. Sie verhindert eine Ausführung der Übersetzung des Kindoperators und gibt einzig $\mathcal{E}$ weiter wie es aus der Erstübersetzung entstand. Das Nachschlagen von id in env ergibt nun die benötigte Variablenumgebung $\mathcal{E}$. Die Übersetzung von $Y_{i,op}$ ergibt nun $\mathcal{E}$, keinen Wert für $\mathcal{Q}$ und die unveränderte Fragmentliste $\mathcal{F}$.

$$\frac{(\text{if } env(id) \text{ is } \mathcal{E})}{\mathcal{F} \vdash Y_{i,op}(v) \Rrightarrow (\mathcal{E}, \emptyset, \mathcal{F})} \tag{5.3}$$

5.3 Tabelle a|b

Der Tabellenoperator erstellt eine neue Tabelle, die aus beliebig vielen Spalten bestehen kann. Die Werte der Spalten sind Information des Operators selbst, er bestitzt keine Kinder. Zur Handhabung in kdb+ betrachten wir entsprechend der Spaltenorientierung eine Tabelle als eine Liste von Listen. Es wird daher für jede Spalte der Tabelle eine eigene Liste erstellt.
Für die Übersetzung sei n die Anzahl der Tupel der neuen Tabelle. Es müssen zwischen den Fällen $m > 1$ und $m = 1$ verschiedene Übersetzungsregeln angewandt werden. Die Elemente der neuen Tabelle sind direkte Informationen des Operators und daher ist diese Fallunterscheidung zur Kompilationszeit durchführbar. Im ersten Falle ist Regel 5.4 anzuwenden. $\mathcal{Q}$ enthält die q Konzepte Listenerstellung und Zuweisung (siehe dazu Abschnitt 3.2).

$$\frac{\mathcal{Q} = \left\{\begin{matrix} \mathrm{q_1} : (x_{11} ; \ldots ; x_{n1}) ; \\ \vdots \quad \vdots \\ \mathrm{q_m} : (x_{1m} ; \ldots ; x_{nm}) ; \end{matrix}\right\}}{\mathcal{F} \vdash \begin{array}{c|c|c} \mathsf{a_1} & \ldots & \mathsf{a_m} \\ \hline x_{11} & \cdots & x_{1m} \\ \vdots & \cdots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{array} \mapsto (\{\mathsf{a_1} \to \mathrm{q_1}, \ldots, \mathsf{a_m} \to \mathrm{q_m}\}, \mathcal{Q}, \mathcal{F})} \tag{5.4}$$

Für Tabellen mit einem zu erstellenden Tupel muss sichergestellt sein, dass durch den Operator a|b wirklich Listen den Ergebnisvariablen zugewiesen werden. Dazu muss die q Funktion `enlist` auf jedes Element des einzigen Tupels angewandt werden wie in Regel 5.5.

$$\frac{\mathcal{Q} = \left\{\begin{matrix} \mathrm{q_1} : \underline{\texttt{enlist}}\ x_1 ; \\ \vdots \quad \vdots \\ \mathrm{q_m} : \underline{\texttt{enlist}}\ x_m ; \end{matrix}\right\}}{\mathcal{F} \vdash \begin{array}{c|c|c} \mathsf{a_1} & \ldots & \mathsf{a_m} \\ \hline x_1 & \cdots & x_m \end{array} \mapsto (\{\mathsf{a_1} \to \mathrm{q_1}, \ldots, \mathsf{a_m} \to \mathrm{q_m}\}, \mathcal{Q}, \mathcal{F})} \tag{5.5}$$

In beiden Fällen sieht man die Variablenumgebung ($\mathcal{E}$) in Aktion. Entsprechend der Benennung der Spaltennamen durch die Algebra $\mathsf{a_1}, \ldots, \mathsf{a_m}$ werden Abbildungen von diesen auf die Variablennamen in q $\mathrm{q_1}, \ldots, \mathrm{q_m}$ in $\mathcal{E}$ eingefügt. In $\mathcal{Q}$ werden diesen Variablen die Ergebniswerte des Operators zugewiesen.

5.4 Attach $@_{\mathsf{a}:v}$

$@_{\mathsf{a}:v}$ fügt an die Eingabelisten eine neue Liste der gleichen Länge mit konstantem Wert v hinzu. Zur Bestimmung der Länge wählen wir beliebig eine Spalte x aus dem Environment des Kindoperators aus. Alle Elemente der neuen Liste sind fixiert auf val. Hierzu benutzen wir die *take* - Funktion von q `(#)`, die für einen Skalarwert als zweites Argument eine Liste der Länge ihres ersten Argumentes erstellt.

$$\frac{\mathcal{F} \vdash \triangle_v \mapsto (\mathcal{E}_v : \{\ldots, \mathsf{x} \to \mathrm{q_x}, \ldots\}, \mathcal{Q}_v, \mathcal{F}_v) \qquad \mathcal{Q} = \{\ \mathrm{q_{res}} : \underline{\texttt{count}}\,[\mathrm{q_x}]\#val ;\ \}}{\mathcal{F} \vdash @_{\mathsf{res}:val}(\triangle_v) \mapsto (\mathcal{E}_v \cup \{\mathsf{res} \to \mathrm{q_{res}}\}, \mathcal{Q}_v \parallel \mathcal{Q}, \mathcal{F}_v)} \tag{5.6}$$

5.5 Tupelfunktionen & Boole'sche Vergleiche

Mehrere Funktionen verbergen sich hinter dem Begriff Tupelfunktion. Auf der einen Seite sind dies arithmetische Funktionen und boole'sche Vergleiche, auf der anderen Funktionen auf Strings. Gemeinsam besitzen sie die Eigenschaft auf jeweils einzelnen Tupeln zu operieren. Bei der Listendarstellung in q wird ein Tupel durch Elemente gleicher Positionsindizes mehrerer Listen repräsentiert.

Es muss zwischen einstelligen und zweistelligen Funktionen bei der Übersetzung unterschieden werden. Zweistellige arithmetische Funktionen wie $+$, $-$, $*$, $/$ (letzteres ist in q: %) und *mod* sowie die zweistelligen boole'schen Vergleiche $>$ und $=$ können direkt analog zu Regel 5.7 umgesetzt werden. In Regel 5.7 ist $+$ zur Veranschaulichung als Muster gewählt. Dabei wird eine neue Spalte res durch den Operator erzeugt, welche das Ergebnis der tupelweisen Addition beinhaltet.

$$\frac{\mathcal{F} \vdash v \Rightarrow (\mathcal{E}_v : \{\mathsf{item}_1 \rightarrow \mathrm{q}_{\mathrm{item1}}, \ldots, \mathsf{item}_2 \rightarrow \mathrm{q}_{\mathrm{item2}}\}, \mathcal{Q}_v, \mathcal{F}_v) \quad \mathcal{Q} = \{\ \mathrm{q}_{\mathrm{res}} : \mathrm{q}_{\mathrm{item1}}\ +\ \mathrm{q}_{\mathrm{item2}}\ ;\ \}}{\mathcal{F} \vdash +_{\mathsf{item}_1,\mathsf{item}_2,\mathsf{res}}(v) \Rightarrow (\mathcal{E}_v \cup \{\mathsf{res} \rightarrow \mathrm{q}_{\mathrm{res}}\}, \mathcal{Q}_v \parallel \mathcal{Q}, \mathcal{F}_v)} \tag{5.7}$$

Für die einstelligen Tupelfunktionen *abs*, *ceiling*, *floor* sowie *not* ist die Übersetzung in Regel 5.8 dargestellt. Exemplarisch wird hier die Übersetzung von *abs* präsentiert.

$$\frac{\mathcal{F} \vdash v \Rightarrow (\mathcal{E}_v : \{\ldots, \mathsf{item} \rightarrow \mathrm{q}_{\mathrm{item}}, \ldots\}, \mathcal{Q}_v, \mathcal{F}_v) \quad \mathcal{Q} = \{\ \mathrm{q}_{\mathrm{res}} : \underline{\mathsf{abs}}\ \mathrm{q}_{\mathrm{item}}\ ;\ \}}{\mathcal{F} \vdash abs_{\mathsf{item},\mathsf{res}}(v) \Rightarrow (\mathcal{E}_v \cup \{\mathsf{res} \rightarrow \mathrm{q}_{\mathrm{res}}\}, \mathcal{Q}_v \parallel \mathcal{Q}, \mathcal{F}_v)} \tag{5.8}$$

Die Pathfinder Algebra umfasst des weiteren die Operationen *fn* : *concat*, *fn* : *contains* und *fn* : *round*. In q stehen für diese keine äquivalenten Operatoren zur Verfügung. Daher müssen für sie separate Übersetzungen realisiert werden.

Für *fn* : *round* gilt die Regel für einstellige Funktionen, wobei $\mathcal{Q}$ wie folgt abzuändern ist.

$$\mathcal{Q} = \{\ \mathrm{q_{res}}\texttt{:}\underline{\texttt{floor}}\ \texttt{0.5+}\ \mathrm{q_{item}}\texttt{;}\quad \}$$

Bei den String-Funktionen ergibt sich zusätzlich die Problematik, dass sämtliche Operationen bezüglich Zeichenketten in q einzig auf dem Datentyp String ausgeführt werden können und nicht auf Symbolen an sich. Da alle Zeichenketten innerhalb der gesamten XQuery-Anfrage - ob in der Dokumententabelle an sich oder in Zwischenergebnissen - als Symbole gespeichert sind, müssen wir diese erst in den Datentyp String und im Falle von *fn* : *concat* nach Abschluss der Operationen wieder zurück in den Datentyp Symbol umwandeln.

Zur Übersetzung wird hierfür die Folgerungsregel 5.7 für zweistellige Funktionen als Ausgangspunkt gewählt und $\mathcal{Q}$ wie folgt angepasst. Für *fn* : *concat*

$$\mathcal{Q} = \left\{\ \mathrm{q_{res}}\texttt{:`\$}\ \underline{\texttt{string}}\texttt{[}\mathrm{q_{item1}}\texttt{],'}\underline{\texttt{string}}\texttt{[}\mathrm{q_{item2}}\texttt{];}\quad \right\}$$

und für *fn* : *contains*

$$\mathcal{Q} = \left\{\ \mathrm{q_{res}}\texttt{:`\$}\ \underline{\texttt{string}}\texttt{[}\mathrm{q_{item1}}\texttt{]}\ \underline{\texttt{like}}\texttt{'}\ \texttt{"*",/:}\underline{\texttt{string}}\texttt{[}\mathrm{q_{item2}}\texttt{],"*";}\ \right\}$$

5.6 Selektion σ_{p}

Der Selektions-Operator σ_{att} entfernt alle Tupel, die in der Spalte att den Wert $false$ beinhalten. In kdb+ wird daher anhand des `where` Operators festgestellt, welche Indizes der korrespondierten Liste zu att den Wert $true$ beinhalten. Diese Indizes werden anschließend auf das komplette Schema angewandt, womit man die Ergebnislisten erhält.

$$\frac{\mathcal{F} \vdash \underset{\sim\sim\sim}{\triangle v} \Rightarrow (\{\mathsf{att} \rightarrow \mathrm{q_{att}}, \mathsf{item_1} \rightarrow \mathrm{q_{item_1}}, \ldots, \mathsf{item_n} \rightarrow \mathrm{q_{item_n}}\}, \mathcal{Q}_v, \mathcal{F}_v) \qquad \mathcal{Q} = \begin{Bmatrix} \texttt{idx:}\underline{\texttt{where}}\ \mathrm{q_{att}}\texttt{;} \\ \mathrm{q_{res_{att}}}\texttt{:}\mathrm{q_{att}}\texttt{[idx];} \\ \mathrm{q_{res_1}}\texttt{:}\mathrm{q_{item_1}}\texttt{[idx];} \\ \vdots \qquad \vdots \\ \mathrm{q_{res_n}}\texttt{:}\mathrm{q_{item_n}}\texttt{[idx];} \end{Bmatrix}}{\mathcal{F} \vdash \overset{\sigma_{\mathsf{att}}}{\underset{\triangle v}{|}} \Rightarrow (\{\mathsf{att} \rightarrow \mathrm{q_{res_{att}}}, \mathsf{item_1} \rightarrow \mathrm{q_{res_1}}, \ldots, \mathsf{item_n} \rightarrow \mathrm{q_{res_n}}\}, \mathcal{Q}_v \parallel \mathcal{Q}, \mathcal{F}_v)} \tag{5.9}$$

5.7 Projektion $\pi_{\mathsf{a_1:b_1,\ldots,a_n:b_n}}$

$$\frac{\mathcal{F} \vdash v \Rightarrow (\{\ldots, \mathsf{b_1} \rightarrow \mathrm{q_1}, \ldots, \mathsf{b_n} \rightarrow \mathrm{q_n}, \ldots\}, \mathcal{Q}, \mathcal{F}_1)}{\mathcal{F} \vdash \pi_{\mathsf{a_1:b_1,\ldots,a_n:b_n}}(v) \Rightarrow (\{\mathsf{a_1} \rightarrow \mathrm{q_1}, \ldots, \mathsf{a_n} \rightarrow \mathrm{q_n}\}, \mathcal{Q}, \mathcal{F}_1)} \quad (5.10)$$

Der Projektions-Operator π lässt sich vollständig zur Kompilationszeit realisieren. Das Entfernen bzw. das Umbenennen der Eingabespalten wird über die Environments des vorhergehenden Operators v und π's geregelt. Dabei sucht π die Q Variablen $\mathrm{q_1}, \ldots, \mathrm{q_n}$ im Environment von v auf und fügt diese entsprechend der neuen Spaltennamen $\mathsf{b_1}, \ldots, \mathsf{b_n}$ in seines hinzu.

5.8 Aggregationen

Für die Aggregationen, die von der Pathfinder Algebra umfasst werden, stehen äquivalente Funktionen in q zur Verfügung. Es sind zwei Fälle der Aggregationen zu unterscheiden, abhängig davon ob eine Partitionierungsattribut definiert ist. Folgende Regel übersetzt Aggregationen ohne Partitionierung.

$$\frac{\mathcal{F} \vdash v \Rightarrow (\{\ldots, \mathsf{item} \rightarrow \mathrm{q_{item}}, \ldots\}, \mathcal{Q}_v, \mathcal{F}_v) \qquad \mathcal{Q} = \{\ \mathrm{q_{res}}\mathtt{:}\underline{\mathtt{sum}}\ \mathrm{q_{item}}\mathtt{;}\ \}}{\mathcal{F} \vdash sum_{\mathsf{item,res}}(v) \Rightarrow (\{\mathsf{res} \rightarrow \mathrm{q_{res}}\}, \mathcal{Q}_v \parallel \mathcal{Q}, \mathcal{F}_v)} \quad (5.11)$$

Regel 5.11 basiert auf der Aggragation *sum*. Analog können die übrigen Aggregationen anhand der q Funktionen `count`, `avg`, `max` und `min` realisiert werden.
Die Aggregation `count`, anhand derer die Kardinalität der Eingabelisten berechnet wird, bildet in diesem Fall eine Ausnahme. Bei ihr ist keine item Spalte explizit angegeben und es bedarf daher der Wahl einer beliebigen Eingabespalte aus der Variablenumgebung des vorhergehenden Operators.

Soweit ein Partitionsattribut definiert ist, werden die Eingabelisten gruppiert und auf den einzelnen Gruppen bzw. Partitionenen die Aggregation unabhängig voneinander durchgeführt.

$$\frac{\mathcal{F} \vdash \triangle_v \Rightarrow (\{\ldots, \mathsf{partition} \to q_{\text{partition}}, \mathsf{item} \to q_{\text{item}}, \ldots\}, \mathcal{Q}_v, \mathcal{F}_v) \qquad \mathcal{Q} = \left\{ \begin{array}{l} \texttt{grpByPartition:}\underline{\texttt{group}}\ q_{\text{partition}}\texttt{;} \\ q_{\text{part_res}}\texttt{:}\underline{\texttt{key}}\ \texttt{grpByPartition;} \\ q_{\text{res}}\texttt{:}\underline{\texttt{sum}}\ \underline{\texttt{each}}\ q_{\text{item}}\texttt{[}\underline{\texttt{value}}\ \texttt{grpByPartition];} \end{array} \right\}}{\mathcal{F} \vdash sum_{\mathsf{partition},\ \mathsf{item,res}}(\triangle_v) \Rightarrow \left(\{\mathsf{res} \to q_{\text{res}}, \mathsf{partition} \to q_{\text{part_res}}\}, \mathcal{Q}_v \parallel \mathcal{Q}, \mathcal{F}_v\right)} \quad (5.12)$$

Wir führen `group` für die Partitionsliste $q_{\text{partition}}$ aus (siehe dazu auch Abschnitt 3.2). Die Domäne des entstandenen Wörterbuches ist zugleich das Ergebnis für $q_{\text{part_res}}$. Der Wertebereich stellt eine zweidimensionale Liste von Indizes der Elemente jeder Partition dar. Die komplexe Liste wird als Index für q_{item} benutzt wie bereits in Listing 3.3 demonstriert wurde. Dadurch werden die Indizes durch die Elemente der Listen ersetzt. Die Aggregationsfunktion `sum` operiert durch `each` schließlich auf jeder dieser Partitionslisten, was Ergebnis für q_{res} zugewiesen wird.

5.9 Joins

Pathfinder unterscheidet zwischen Equi-Joins, Semi-Joins und Theta-Jons. Für keine dieser Varianten können eingebaute q Join-Funktionen verwendet werden. Dadurch entstehen unterschiedliche Übersetzungsregeln für die verschiedenen Varianten.

5.9.1 Equi-Join $\bowtie_{\mathsf{join1=join2}}$

Joins in kdb+ sind besonders darauf ausgelegt, effizient auf Tabellen zu operieren, von denen eine Tabelle Fremdschlüsselreferenzen auf die andere besitzt. q bietet ausschließlich Join-Funktion, bei denen mindestens eine der Spalten des Verbundkriteriums eine Schlüsselspalte ist. Daher müssen wir uns für den Operator $\bowtie_{a=b}$ eigene Funktionen definieren. Wir definieren die Hilfsfunktion `eqJoin`, die mit Basisfunktionen von q den Vergleich der Kriteriumsspalten durchführt und die Indizes der zu verbindenden Elemente zurückgibt.

Listing 5.1: *Iterativer* Equi-Join

```
1 eqJoin:{[leftCol;rightCol];
2  : {:where x=rightCol} each leftCol;
3  };
```

`eqJoin` gibt für jedes Element der linken Liste die Positionen der Elemente mit gleichem Wert in der rechten Liste zurück. `{:where x=rightCol}` ist dabei eine *inline* definierte Funktion. `x` ist der implizite Parameter der Funktion, in diesem Falle je Aufruf ein Element aus `leftCol`. Ohne Unterfunktion ließe sich dieser Vorgang kürzer und einfacher wie folgt formulieren:

```
1 where each leftCol =/: rightCol
```

. Die hier vorgestellte, iterative Variante bietet jedoch den Vorteil, dass der Speicherbedarf im Normalfall wesentlich reduziert ist, da fehlgeschlagene Vergleiche sofort durch `where` abgestoßen werden. Die Anwendung von `leftCol =/: rightCol` führt unumgänglich zu einem Zwischenergebnis der Länge von `leftCol` multipliziert mit der Länge von `rightCol`, was die Funktion bei großen Eingabelisten sehr verlangsamt.

In Regel 5.13 ist der gesamte Ablauf von $\bowtie_{join1=join2}$ dargestellt.

$$\frac{\begin{array}{c}\mathcal{F} \vdash \triangle_{v} \Rightarrow (\mathcal{E}_v, \mathcal{Q}_v, \mathcal{F}_v) \\ \mathcal{E}_v = \{\mathsf{join_1} \rightarrow \mathrm{q_{join_1}}, \mathsf{item_{v1}} \rightarrow \mathrm{q_{item_{v1}}}, \ldots, \mathsf{item_{vn}} \rightarrow \mathrm{q_{item_{vn}}}\} \\ \mathcal{F}_v \vdash \triangle_{w} \Rightarrow (\mathcal{E}_w, \mathcal{Q}_w, \mathcal{F}_w) \\ \mathcal{E}_w = \{\mathsf{join_2} \rightarrow \mathrm{q_{join_2}}, \mathsf{item_{w1}} \rightarrow \mathrm{q_{item_{w1}}}, \ldots, \mathsf{item_{wn}} \rightarrow \mathrm{q_{item_{wn}}}\} \\ \mathcal{Q} = \left\{\begin{array}{l} \texttt{joinIdx:eqJoin[}\mathrm{q_{join1}}\texttt{;}\mathrm{q_{join2}}\texttt{];} \\ \texttt{leftIdx:leftIndexes[joinIdx];} \\ \texttt{rightIdx:\underline{raze} joinIdx;} \\ \mathrm{q_{v0}}\texttt{:}\mathrm{q_{join1}}\texttt{[leftIdx];} \\ \mathrm{q_{v1}}\texttt{:}\mathrm{q_{item_{v1}}}\texttt{[leftIdx];} \\ \quad\vdots \\ \mathrm{q_{vn}}\texttt{:}\mathrm{q_{item_{vn}}}\texttt{[leftIdx];} \\ \mathrm{q_{w0}}\texttt{:}\mathrm{q_{join2}}\texttt{[rightIdx];} \\ \mathrm{q_{w1}}\texttt{:}\mathrm{q_{item_{w1}}}\texttt{[rightIdx];} \\ \quad\vdots \\ \mathrm{q_{wn}}\texttt{:}\mathrm{q_{item_{wn}}}\texttt{[rightIdx];} \end{array}\right\} \\ \mathcal{E} = \{\mathsf{join_1} \rightarrow \mathrm{q_{v0}}, \mathsf{item_{v1}} \rightarrow \mathrm{q_{v1}}, \ldots, \mathsf{item_{vn}} \rightarrow \mathrm{q_{item_{vn}}}\} \cup \\ \{\mathsf{join_2} \rightarrow \mathrm{q_{w0}}, \mathsf{item_{w1}} \rightarrow \mathrm{q_{w1}}, \ldots, \mathsf{item_{wn}} \rightarrow \mathrm{q_{item_{wn}}}\} \end{array}}{\mathcal{F} \vdash \bowtie_{\mathsf{join_1=join_2}}(\triangle_{v}, \triangle_{w}) \Rightarrow (\mathcal{E}, \mathcal{Q}_v \parallel \mathcal{Q}_w \parallel \mathcal{Q}, \mathcal{F}_w)} \quad (5.13)$$

Nachdem `eqJoin` mit beiden Joinkriterien als Paramter aufgerufen wurde, verfügt man über eine Liste von Listen mit Indizes der rechten Eingabelisten (`joinIdx`). Alle Elemente, die durch die Indizes der ersten Liste aus `joinIdx` referenziert sind, werden mit den ersten Elementen der linken Eingabelisten verbunden, die der zweiten Liste mit den zweiten Elementen etc.
Funktion `leftIdx` vervielfältigt die Indizes der linken Eingabetabellen, um ausreichend häufig in übereinstimmender Anzahl zu ihren zukünftigen Gegenstücken der rechten Eingabelisten vorhanden zu sein.

Listing 5.2: Funktion `leftIndexes`

```
1 leftIndexes:{[idx];
2  :raze (count each idx)#'key count idx;
3  };
```

Um die Indizes der zukünftigen rechten Listen zu erhalten, ist es ausreichend, die Liste `joinIdx` mit `raze` abzuflachen. Man verfügt nun mit `leftIdx` und `rightIdx` über die Indizes der resultierenden Listen. Diese werden abschließend auf die entsprechenden Eingabelisten angewandt und ergeben damit die Ergebnislisten.

Gruppierender Equi-Join

Für die Implementierung der Funktion `eqJoin` wird nun eine Optimierung der soeben vorgestellten Version gezeigt. Auf der Suche nach einer Möglichkeit auf jegliche Iterationen durch Adverbien wie `each` zu verzichten, entstand folgende Implementierung.

Listing 5.3: *Gruppierender* Equi-Join

```
1 eqJoin:{[leftCol;rightCol];
2  :group[rightCol][leftCol];
3  };
```

In dieser Version gruppiert die Funktion `eqJoin` die Join-Spalte der rechten Tabelle (siehe Abschnitt 3.2) und wendet auf das erhaltene *Dictionary* die Werte der linken Join-Spalte an. Das Ergebnis ist äquivalent zu der *iterativen Equi-Join*-Variante, stellt sich jedoch besonders bei großen Eingabelisten als vorteilhafter heraus. Ein Vergleich beider Varianten wird im Rahmen der Experimente in Kapitel 6.2 vorgestellt.

5.9.2 Semi-Join $\ltimes_{a=b}$

$\ltimes_{\mathsf{join}_1=\mathsf{join}_2}$ prüft für Elemente des ersten Joinkriteriums join_1, ob sie im Joinkriterium des zweiten Kindoperators join_2 vorkommen. Wird ein Vorkommen

gefunden, so wird das Tupel der linken Eingabelisten in das Ergebnis übernommen.

$$
\frac{
\begin{array}{c}
\mathcal{F} \vdash v \Rightarrow (\mathcal{E}_v, \mathcal{Q}_v, \mathcal{F}_v) \\
\mathcal{E}_v = \{\mathsf{join_1} \rightarrow \mathrm{q_{join_1}}, \mathsf{item_{v1}} \rightarrow \mathrm{q_{item_{v1}}}, \ldots, \mathsf{item_{vn}} \rightarrow \mathrm{q_{item_{vn}}}\} \\
\mathcal{F}_v \vdash w \Rightarrow (\{\mathsf{join_2} \rightarrow \mathrm{q_{join_2}}\}, \mathcal{Q}_w, \mathcal{F}_w) \\
\mathcal{Q} = \left\{ \begin{array}{l}
\texttt{idx:}\underline{\texttt{where}}\ \mathrm{q_{join_1}}\ \underline{\texttt{in}}\ \mathrm{q_{join_2}}\texttt{;} \\
\mathrm{q_0}\texttt{:}\mathrm{q_{join_1}}\texttt{[idx];} \\
\mathrm{q_1}\texttt{:}\mathrm{q_{item_{v1}}}\texttt{[idx];} \\
\quad\vdots \\
\mathrm{q_n}\texttt{:}\mathrm{q_{item_{vn}}}\texttt{[idx];}
\end{array} \right\} \\
\mathcal{E} = \{\mathsf{join_1} \rightarrow \mathrm{q_0}, \mathsf{item_{v1}} \rightarrow \mathrm{q_1}, \ldots, \mathsf{item_{vn}} \rightarrow \mathrm{q_n}\}
\end{array}
}{
\mathcal{F} \vdash \ltimes_{\mathsf{join_1}=\mathsf{join_2}}(v, w) \Rightarrow (\mathcal{E}, \mathcal{Q}_v \parallel \mathcal{Q}_w \parallel \mathcal{Q}, \mathcal{F}_w)
} \tag{5.14}
$$

Die Prüfung welche Elemente einer Liste in einer anderen Liste vorkommen, geschieht mit der q Funktion `in`. `in` liefert eine boole'sche Liste, die anzeigt welche Tests erfolgreich und welche es nicht waren. Durch `where` werden die Indizes der Elemente geliefert, die den Test bestanden haben (`idx`). Die Indizes werden nun auf alle Eingabelisten des linken Kindoperators angewandt und den Ergebislisten zugeordnet.

5.9.3 Theta-Join $\bowtie_{v\theta w}$

$\bowtie_{v\theta w}$ stellt eine oder mehrere Joinbedingungen an die Eingaberelationen. Die erste Bedingung wird ähnlich der iterativen Variante eines Equi-Join (5.9.1) behandelt. Funktion `leftIndexes` kann direkt aus Listing 5.2 für $\bowtie_{a\theta b}$ übernommen werden. Regel 5.15 beschreibt die Übersetzung des Operators Theta-Join für $\bowtie_{\substack{\mathsf{item}_{v_1} \leq \mathsf{item}_{w_1}, \\ \mathsf{item}_{v_2} = \mathsf{item}_{w_2}}}$.

$$
\frac{
\begin{array}{c}
\mathcal{F} \vdash \triangle_{v} \Rrightarrow (\mathcal{E}_v, \mathcal{Q}_v, \mathcal{F}_v) \\
\mathcal{E}_v = \{\mathsf{item}_{\mathsf{v}_1} \rightarrow \mathrm{q}_{\mathrm{item}_{\mathrm{v}_1}}, \ldots, \mathsf{item}_{\mathsf{v}_\mathsf{m}} \rightarrow \mathrm{q}_{\mathrm{item}_{\mathrm{v}_\mathrm{m}}}\} \\
\mathcal{F}_v \vdash \triangle_{w} \Rrightarrow (\mathcal{E}_w, \mathcal{Q}_w, \mathcal{F}_w) \\
\mathcal{E}_w = \{\mathsf{item}_{\mathsf{w}_1} \rightarrow \mathrm{q}_{\mathrm{item}_{\mathrm{w}_1}}, \ldots, \mathsf{item}_{\mathsf{w}_\mathsf{n}} \rightarrow \mathrm{q}_{\mathrm{item}_{\mathrm{w}_\mathrm{n}}}\} \\
\mathcal{Q}_1 = \left\{ \begin{array}{l}
\texttt{idx:}\underline{\texttt{where}}\texttt{\{x <= } \mathrm{q}_{\mathrm{item}_{\mathrm{w}_1}}\texttt{\} } \underline{\texttt{each}}\ \mathrm{q}_{\mathrm{item}_{\mathrm{v}_1}}\texttt{;} \\
\texttt{left:leftIndexes[idx];} \\
\texttt{right:}\underline{\texttt{raze}}\texttt{ idx;}
\end{array} \right\} \\
\mathcal{Q}_2 = \left\{ \begin{array}{l}
\texttt{idx:}\underline{\texttt{where}}\ \mathrm{q}_{\mathrm{item}_{\mathrm{v}_2}}\texttt{[left]=}\mathrm{q}_{\mathrm{item}_{\mathrm{w}_2}}\texttt{[right];} \\
\texttt{left:left[idx];} \\
\texttt{right:right[idx];}
\end{array} \right\} \\
\mathcal{Q}_3 = \left\{ \begin{array}{l}
\mathrm{q}_{\mathrm{v}_1}\texttt{:}\mathrm{q}_{\mathrm{item}_{\mathrm{v}_1}}\texttt{[left];} \\
\ \vdots \quad \vdots \\
\mathrm{q}_{\mathrm{v}_\mathrm{m}}\texttt{:}\mathrm{q}_{\mathrm{item}_{\mathrm{v}_\mathrm{m}}}\texttt{[left];} \\
\mathrm{q}_{\mathrm{w}_1}\texttt{:}\mathrm{q}_{\mathrm{item}_{\mathrm{w}_1}}\texttt{[right];} \\
\ \vdots \quad \vdots \\
\mathrm{q}_{\mathrm{w}_\mathrm{n}}\texttt{:}\mathrm{q}_{\mathrm{item}_{\mathrm{w}_\mathrm{n}}}\texttt{[right];}
\end{array} \right\} \\
\mathcal{Q} = \{\mathcal{Q}_1 \parallel \mathcal{Q}_2 \parallel \mathcal{Q}_3\} \\
\mathcal{E} = \{\mathsf{item}_{\mathsf{v}_1} \rightarrow \mathrm{q}_{\mathrm{v}_1}, \ldots, \mathsf{item}_{\mathsf{v}_\mathsf{m}} \rightarrow \mathrm{q}_{\mathrm{v}_\mathrm{m}}\} \cup \\
\{\mathsf{item}_{\mathsf{w}_1} \rightarrow \mathrm{q}_{\mathrm{w}_1}, \ldots, \mathsf{item}_{\mathsf{w}_\mathsf{n}} \rightarrow \mathrm{q}_{\mathrm{w}_\mathrm{n}}\}
\end{array}
}{
\mathcal{F} \vdash \bowtie_{\substack{\mathsf{item}_{v_1} \leq \mathsf{item}_{w_1} \\ \mathsf{item}_{v_2} = \mathsf{item}_{w_2}}} (\triangle_{v}, \triangle_{w}) \Rrightarrow (\mathcal{E}, \mathcal{Q}_v \parallel \mathcal{Q}_w \parallel \mathcal{Q}, \mathcal{F}_w)
}
\tag{5.15}
$$

Bei allen weiteren Joinbedingungen wird zuerst durch Indizierung mit `left` bzw. `right` auf die Elemente beschränkt, die aufgrund der vorhergegangen Vergleiche als bislang gültig anzusehen sind. Diese Teilmenge der Eingabelisten wird dann tupelweise verglichen. Die Indizes `left` und `right` werden nach diesem Vergleich auf Elemente mit wahrem Vergleichsergebnis reduziert. Die Regel zeigt zwei Verbundskriterien. Sind mehrere gegeben, so wird $\mathcal{Q}_2$ entsprechend häufig mit angepassten Vergleichswerten wiederholt. Sind alle Vergleiche durchgeführt, werden die Eingeabelisten mit `left` bzw. `right` indiziert und ergeben das Ergebnis.
Für den Fall, dass der erste Vergleich einem Equi-Joins entspricht, ist die entsprechende Funktion des *gruppierenden* Equi-Join aus Listing 5.3 in $\mathcal{Q}_1$ anzuwenden.

5.10 Duplikateliminierung δ

Die Duplikateliminierung betrachtet alle Elemente eines Tupels. Es ist daher notwendig, einen Bezug zwischen den einzelnen Elementen der Eingabelisten herzustellen. Dafür bieten sich zwei Varianten an:

1. Erstellen einer Liste aus Tupeln der Art $(x_{i0}, \ldots, x_{in})$, wobei x_{ij} das Element an Position j der i-ten Eingabeliste ist.
2. Erstellen einer Tabelle mit den Eingabelisten als Spalteninhalte.

Die erste Transformation lässt sich durch Anwendung der elementweisen Konkatenation `(,')`, die zweite Variante direkt mit der q Syntax zur Erstellung von Tabellen verwirklichen. Wir wählen die Tabelle als Datenstruktur für die Duplikatelimination, da das Aufsetzen der Tabellen und die eigentliche Operation der Duplikatselimination auf einer Tabelle Geschwindigkeitsvorteile zeigt.
Zur Überführung der Listen in eine Tabellendarstellung definieren wir die Funktion `qTable`:

Listing 5.4: Funktion *qTable* erstellt eine Tabelle für zwei Listen mit Spaltenname und -werte

```
1 qTable:{[names;values];
2  :flip names!values;
3  };
```

`qTable` erwartet die Listen `names` als Liste von Spaltennamen der zu erstellenden Tabelle und `values` als Liste von Spalteninhalten.

$$\frac{\mathcal{F} \vdash \underset{\sim\sim\sim}{\triangle v} \mapsto (\{\mathsf{item_1} \to q_{\mathrm{item_1}}, \ldots, \mathsf{item_n} \to q_{\mathrm{item_n}}\}, \mathcal{Q}_v, \mathcal{F}_v) \quad \mathcal{Q} = \left\{\begin{array}{l} \texttt{itemNames:`}\mathsf{item_1}\texttt{...`}\mathsf{item_n}\texttt{;} \\ \texttt{itemValues:(}q_{\mathrm{item_1}}\texttt{;...;}q_{\mathrm{item_n}}\texttt{);} \\ \texttt{tbl:qTable[itemNames;itemValues];} \\ \texttt{res_lst:\underline{value} \underline{flip} \underline{distinct} tbl;} \\ q_{\mathrm{res_1}}\texttt{:res_lst[0];} \\ \quad \vdots \qquad\qquad \vdots \\ q_{\mathrm{res_n}}\texttt{:res_lst[n-1];} \end{array}\right\}}{\mathcal{F} \vdash \overset{\delta}{\underset{\sim\sim\sim}{\triangle v}} \mapsto (\{\mathsf{item_1} \to q_{\mathrm{res_1}}, \ldots, \mathsf{item_n} \to q_{\mathrm{res_n}}, \mathcal{Q}_v \parallel \mathcal{Q}, \mathcal{F}_v)} \quad (5.16)$$

`itemNames` als erster Parameter für `qTable` muss hierbei gesetzt werden entsprechend den Eingabelisten von δ. Bei $n = 3$ nehmen wir als Eingabe

die Listen item_1, item_2 und item_3 und die korrespondierenden q Variablen aus dem Environment des vorhergehenden Operators $\mathrm{q}_{\mathrm{item}_1}$, $\mathrm{q}_{\mathrm{item}_2}$ und $\mathrm{q}_{\mathrm{item}_3}$ an. Sonderfall für $n = 1$: Auf die dann einzigen Elemente `item₁` und $\mathrm{q}_{\mathrm{item}_1}$ muss jeweils vor Verwendung `enlist` angewandt werden, um die Listenform zu generieren.

5.11 Mengenoperationen $\setminus$, $\cap$, $\dot\cup$

Mengenoperationen operieren auf den Eingabelisten von Eingabeoperatoren, die jeweils gleiche Bezeichnungen der Listen besitzen. Es müssen hier nun ganze Tupel betrachtet werden. Um Zusammenhänge zwischen den einzelnen Einträgen der Eingabelisten eines Kindoperators herstellen zu können, verwenden wir hier wie schon bei der Duplikatelimination δ Tabellen, die als Spaltennamen die Namen der Algebravariablen führen. Dies geschieht auch in diesem Fall anhand der in Abschnitt 5.10 definierten Funktion Funktion `qTable`. Bestehen nun für beide Eingabeoperatoren jeweils eine Tabelle mit den Eingabewerten, so kann q direkt auf den Tupeln dieser Tabellen arbeiten. Für die Mengendifferenz $\setminus$ eignet sich das semantische Äquivalent in q, die `except` Funktion. Als Ergebnis von `except` erhält man eine Tabelle, die lediglich die Einträge der Tabelle mit den Werten des linken Kindes erhält, die nicht vom rechten Kind geführt wurden. Abschließend sind die einzelnen Spalten der Ergebnistabelle (`res_tbl`) wieder einzelnen Listen zuzuordnen.

$$\frac{\begin{array}{c}\mathcal{F} \vdash \triangle_v \Rrightarrow \left(\left\{\mathsf{item_1} \to \mathrm{q}_{\mathrm{v}_1}, \ldots, \mathsf{item_n} \to \mathrm{q}_{\mathrm{v}_n}\right\}, \mathcal{Q}_v, \mathcal{F}_v\right) \\ \mathcal{F}_v \vdash \triangle_w \Rrightarrow \left(\left\{\mathsf{item_1} \to \mathrm{q}_{\mathrm{w}_1}, \ldots, \mathsf{item_n} \to \mathrm{q}_{\mathrm{w}_n}\right\}, \mathcal{Q}_w, \mathcal{F}_w\right) \\ \mathcal{Q} = \left\{\begin{array}{l}\texttt{itemNames:`item}_1\texttt{...`item}_n\texttt{;} \\ \texttt{leftValues:(}\mathrm{q}_{\mathrm{v}_1}\texttt{;...;}\mathrm{q}_{\mathrm{v}_n}\texttt{);} \\ \texttt{rightValues:(}\mathrm{q}_{\mathrm{w}_1}\texttt{;...;}\mathrm{q}_{\mathrm{w}_n}\texttt{);} \\ \texttt{res_tbl:}\underline{\texttt{except}}\texttt{[qtable[itemNames;leftValues];} \\ \qquad\qquad\texttt{qtable[itemNames;rightValues]];} \\ \mathrm{q}_{\mathrm{res}_1}\texttt{:res_tbl[`item}_1\texttt{];} \\ \quad\vdots \qquad\qquad \vdots \\ \mathrm{q}_{\mathrm{res}_n}\texttt{:res_tbl[`item}_n\texttt{];}\end{array}\right\} \\ \mathcal{E} = \left\{\mathsf{item_1} \to \mathrm{q}_{\mathrm{res}_1}, \ldots, \mathsf{item_n} \to \mathrm{q}_{\mathrm{res}_n}\right\}\end{array}}{\mathcal{F} \vdash \setminus(\triangle_v, \triangle_w) \Rrightarrow \left(\mathcal{E}, \mathcal{Q}_v \parallel \mathcal{Q}_w \parallel \mathcal{Q}, \mathcal{F}_w\right)} \tag{5.17}$$

Die Operationen $\cap$ und $\dot\cup$ können anhand der gleichen Regel gelöst werden. Die entsprechenden q Funktionen sind dann die Schnittmengenfunktion `inter` bzw. die Listenkonkatenation `(,)`.

5.12 Kreuzprodukt $\times$

Die Operation $\times$ lässt sich mit dem semantischen Äquivalent in q, `cross`, ähnlich wie die Mengenoperationen übersetzen. Im Falle von $\times$ besitzen die Eingabelisten jedoch jeweils unterschiedliche Bezeichnungen. Das Schema von $\times$ stellt dazu eine Vereinigung der Schemata beider Kind-Operatoren dar.

$$\frac{\begin{array}{l} \mathcal{F} \vdash \triangle_{v} \mapsto \left(\{\mathsf{v_1} \to \mathrm{q_{v_1}}, \ldots, \mathsf{v_m} \to \mathrm{q_{v_m}}\}, \mathcal{Q}_v, \mathcal{F}_v\right) \\ \mathcal{F}_v \vdash \triangle_{w} \mapsto \left(\{\mathsf{w_1} \to \mathrm{q_{w_1}}, \ldots, \mathsf{w_n} \to \mathrm{q_{w_n}}\}, \mathcal{Q}_w, \mathcal{F}_w\right) \\ \mathcal{Q} = \left\{\begin{array}{l} \texttt{leftNames:`}\mathsf{v_1}\ldots\texttt{`}\mathsf{v_m}\texttt{;} \\ \texttt{leftValues:(}\mathrm{q_{v_1}}\texttt{;}\ldots\texttt{;}\mathrm{q_{v_m}}\texttt{);} \\ \texttt{rightNames:`}\mathsf{w_1}\ldots\texttt{`}\mathsf{w_n} \\ \texttt{rightValues:(}\mathrm{q_{w_1}}\texttt{;}\ldots\texttt{;}\mathrm{q_{w_n}}\texttt{);} \\ \texttt{res_tbl:}\underline{\texttt{cross}}\texttt{[qtable[leftNames;leftValues];} \\ \qquad\qquad\qquad \texttt{qtable[rightNames;rightValues]];} \\ \mathrm{q_{res_{v_1}}}\texttt{:res_tbl[`}\mathsf{v_1}\texttt{];} \\ \quad\vdots \qquad\qquad \vdots \\ \mathrm{q_{res_{v_m}}}\texttt{:res_tbl[`}\mathsf{v_m}\texttt{];} \\ \mathrm{q_{res_{w_1}}}\texttt{:res_tbl[`}\mathsf{w_1}\texttt{];} \\ \quad\vdots \qquad\qquad \vdots \\ \mathrm{q_{res_{w_n}}}\texttt{:res_tbl[`}\mathsf{w_n}\texttt{];} \end{array}\right\} \\ \mathcal{E} = \left\{\mathsf{v_1} \to \mathrm{q_{res_{v_1}}}, \ldots, \mathsf{v_m} \to \mathrm{q_{res_{v_m}}}, \mathsf{w_1} \to \mathrm{q_{res_{w_1}}}, \ldots, \mathsf{w_n} \to \mathrm{q_{res_{w_n}}}\right\} \end{array}}{\mathcal{F} \vdash \times(\triangle_{v}, \triangle_{w}) \mapsto \left(\mathcal{E}, \mathcal{Q}_v \parallel \mathcal{Q}_w \parallel \mathcal{Q}, \mathcal{F}_w\right)} \tag{5.18}$$

Wie auch für die Mengenoperatoren aus Abschnitt 5.11 werden für die Eingabelisten beider Eingabeoperatoren jeweils eine Tabelle erzeugt, wobei die Spaltennamen den Namen der Eingabelisten entsprechen. Die Anwendung von `cross` auf beiden Tabellen ist auch analog zu der Übersetzung der Mengenoperationen. Abschließend müssen alle Spalten entsprechend ihrer Namen den Ergebnislisten zugewiesen werden.

5.13 Tupelnummerierung $\rho_{<\mathsf{a_1},\ldots,\mathsf{a_n}>,\mathsf{b},\mathsf{p}}$

Der Operator $\rho_{\langle \mathsf{item_1}[asc/desc],\ldots,\mathsf{item_n}[asc/desc]\rangle,\mathsf{res},\mathsf{partition}}$ (Tupelnummerierung) zieht für die Nummerierung ein Partitionierungsattribut partition in Betracht. Grundsätzlich werden die Aufzählungswerte anhand von Sortierkriterien innerhalb jeder Partition unabhängig erzeugt.

Als Sortierkriterien können mehrere Listen und diese in entweder auf- oder absteigender Folge betrachtet werden. Um mehrere Listen gleichzeitig zu betrachten, erstellt man eine Liste von Tupeln bestehend aus jeweils einem Element jeder Eingabeliste. Eine solch geartete Liste erhält man durch mehrfaches Anwenden von *concatenate-each* (siehe für *each*-Operator Abschnitt 3.2).
Bei Listen wird vor der Konkatenation `reverse` angewandt, wenn diese in absteigender Reihenfolge in Betracht zu ziehen sind. `reverse` bewirkt, dass die Liste im Argument in umgekehrter Reihenfolge zurückgegeben wird.
Die erhaltene und den Sortierkriterien entsprechende Liste wird der Variablen `sortList` in folgender Regel 5.19 zugewiesen.

$$\frac{\begin{array}{c}\mathcal{F}_v \vdash v \Rightarrow (\mathcal{E}, \mathcal{Q}_v, \mathcal{F}_v) \\ \mathcal{E}_v = \{\ldots, \mathsf{partition} \rightarrow \mathrm{q_{partition}}, \ldots, \mathsf{item_1} \rightarrow \mathrm{q_{item_1}}, \ldots, \mathsf{item_n} \rightarrow \mathrm{q_{item_n}}, \ldots\} \\ \mathcal{Q} = \left\{\begin{array}{l} \texttt{sortList:([}\underline{\texttt{reverse}}\texttt{]}\ \mathrm{q_{item_1}}\texttt{),'...,'([}\underline{\texttt{reverse}}\texttt{]}\ \mathrm{q_{item_n}}\texttt{);} \\ \texttt{partitionsGrouped:}\underline{\texttt{value}}\ \underline{\texttt{group}}\ \mathrm{q_{partition}}\texttt{;} \\ \texttt{partitionsVals:sortList[partitionsGrouped];} \\ \texttt{partitionsRanked:1+}\underline{\texttt{raze}}\ \underline{\texttt{rank}}\ \underline{\texttt{each}}\ \texttt{partitionsVals;} \\ \mathrm{q_{res}}\texttt{:partitionsRanked[}\underline{\texttt{iasc}}\ \underline{\texttt{raze}}\ \texttt{partitionsGrouped];} \end{array}\right\}\end{array}}{\mathcal{F} \vdash \rho_{<\mathsf{item_1}[asc/desc],\ldots,\mathsf{item_n}[asc/desc]>,\mathsf{res},\mathsf{partition}}(v) \Rightarrow (\mathcal{E}_v \cup \{\mathsf{res} \rightarrow \mathrm{q_{res}}\}, \mathcal{Q}_v \parallel \mathcal{Q}, \mathcal{F}_v)} \tag{5.19}$$

Partitionen werden erstellt durch Gruppierung mit `group` des Partitionierungsattributes. Der Wertebereich des dadurch erhaltenen *Dictionary* besteht aus Listen, die für jede Partition die Indizes der Elemente führen (`partitionGrouped`).
Im nächsten Schritt werden die Sortierkriteriums `sortedList` zusammengesetzt und durch die Indizes aus `partitionGrouped` ihren Partitionen zugeordnet. Auf jeder dieser einzelnen Listen wird nun eine Ordnungsnummer mit `rank each` generiert. Die entstandene Liste von Listen wird abgeflacht und jeder Nummerierungswert noch um eins erhöht, um eine Nummerierung beginnend bei eins zu gewährleisten. Um nun von dieser Partitionsdarstellung

aus die erlangten Werte den wirklichen Werten bzw. Positionen in der Liste zuzuordnen, benutzen wir `iasc` um die Gruppierung wieder zu invertieren. Zur Verdeutlichung kann man sich `partitionsGrouped` als Indexreihenfolge vorstellen, die eine Eingabeliste entsprechend eines Partitionskriterium gruppiert. Mit `iasc` erhält man die Indizes der Elemente in der Reihenfolge, die notwendig ist, um die ursprünglichen Positionen der Elemente wieder herzustellen und die Gruppierung *aufzulösen*. Die generierten Nummerierungswerte werden so ihren korrespondierenden Elementen in den Listen zugeordnet.

5.14 Typumwandlung $cast_{type,n}$

Funktion $cast_{type,n}$ führt eine möglicherweise polymorphe Eingabeliste n in einen Zieldatentyp (*type*) über. Wir wollen die Übersetzung in den Zieldatentyp *int* in Regel 5.20 darstellen.

$$\frac{\mathcal{F} \vdash \overset{v}{\triangle} \Rrightarrow (\mathcal{E}_v : \{\ldots, \mathsf{item} \to \mathrm{q}_{\mathrm{item}}, \ldots\}, \mathcal{Q}_v, \mathcal{F}_v) \qquad \mathcal{Q} = \{\ \mathrm{q}_{\mathrm{res}}\texttt{:7h\$}\mathrm{q}_{\mathrm{item}}\texttt{;}\ \ \}}{\mathcal{F} \vdash \overset{cast_{int,\mathsf{item},\mathsf{res}}}{\underset{v}{\triangle}} \Rrightarrow (\{\mathcal{E}_v \cup \mathsf{res} \to \mathrm{q}_{\mathrm{res}}\}, \mathcal{Q}_v \parallel \mathcal{Q}, \mathcal{F}_v)} \quad (5.20)$$

Es wird dabei wie in der Einführung zu q in Kapitel 3.2 der *Casting*-Operator von q (`$`) auf die Eingabeliste angewendet für einen definierten Zieldatentyp. Zu Pathfinder-Zieldatentyp *int* korrespondiert der q Datentyp *long*. Die numerische Darstellung von *long*, die wir für (`$`) benutzen entspricht `7h`. Für alle Datentypen und deren numerischen Identifikationen sei auf die q Sprachenreferenz verwiesen ([Ort06]).
Es gilt zur Kompilationszeit zwei Ausnahmen zu beachten. Für die Umwandlung zu Datentypen, die Symbolen in q korrespondieren, ist eine Abänderung von $\mathcal{Q}$ aus Regel 5.20 notwendig.

$$\mathcal{Q} = \{\ \mathrm{q}_{\mathrm{res}}\texttt{:`\$}\underline{\texttt{string}}\ \mathrm{q}_{\mathrm{item}}\texttt{;}\ \ \}$$

Es wird zuerst jedes Element der Eingabeliste in ein Zeichenfeld mit `string` umgewandelt, bevor diese schließlich mit `` `$ `` in *Symbole* umgewandelt werden. Darüber hinaus benötigt die Umwandlung von Daten des Types *Symbol* zu Zahlentypen in q eine entsprechende Handhabung.

$$\mathcal{Q} = \{\ \mathrm{q}_{\mathrm{res}}\texttt{:7h\$"F"\$}\underline{\texttt{string}}\ \mathrm{q}_{\mathrm{item}}\texttt{;}\}$$

Hierfür müssen Symbole erneut zuerst in Strings umgewandelt werden. Diese Zeichenketten werden nun durch `"F"` nach q Konvention in Fließkommazahlen umgewandelt, bevor sie schließlich mit erneutem Anwenden von `($)` in ihren Zieldatentyp – hier wieder *int* gewählt – umgewandelt werden. Die Mehrarbeit durch temporäre Umwandlung zu Fließkommazahlen muss in Kauf genommen werden, da eine direkte Umwandlung von beispielsweise Symbolen der Art `‘42.09` in Ganzzahlen fehlschlägt.

5.15 Dokumentensuche $doc - lookup_{\mathsf{p,n}}$

Der Operator $doc - lookup_{\mathsf{p,n}}$ fügt für jeden Partitionswert p ein persistent gespeichertes Dokument zum Gültigkeitsbereich der derzeitigen XQuery-Abfrage hinzu. Er durchsucht alle gespeicherten Dokumente in $\mathcal{D}$ und hängt das mit der zum Eingabewert passenden Bezeichnung an die globale Fragmentliste $\mathcal{F}$.

Wir benötigen eine Hilfsfunktion `newFragments` zum Erstellen einer Liste von Fragmenten bestehend aus den gesuchten Dokumententabellen.

```
1  newFragments:{[documents;item];
2   names:documents .\: (0;‘val);
3   idx:where each item =\: names;
4   :documents[idx];
5  };
```

`newFragments` liest zuerst alle Dokumentenbezeichnungen aus der Liste von Dokumenten `documents` aus. Durch Indizierung in die Tiefe mit dem `(.)`-Operator wird der Wert der Spalte val des Tupels mit Index 0 ausgelesen. Das Adverb `\:` führt diese Operation auf jeder Tabelle in $\mathcal{D}$ aus. Damit sind in `names` alle Bezeichnungen der Dokumententabelle abgelegt. Im nächsten Schritt werden jeweils alle Bezeichnungen der gesuchten Dokumente `item` mit den Namen den Bezeichnungen in `names` verglichen. Man erhält eine Matrix, die für jedes Element in `item` angibt ob es mit den Element in `names` übereinstimmt. Die Anwendung von `where` auf jede Zeile dieser Matrix mit `each` liefert den Index des passenden Dokumentes zu jedem Element in `item`. Die Indizes werden auf alle Dokumente angewendet, was den Rückgabewert der Funktion `newFragments` darstellt.

$$\frac{\begin{array}{l}\mathcal{F} \vdash \underset{\sim\sim\sim}{\triangle v} \mapsto (\{\mathsf{iter} \rightarrow \mathrm{q}_{\mathrm{iter_v}}, \mathsf{item} \rightarrow \mathrm{q}_{\mathrm{item_v}}\}, \mathcal{Q}_v, \mathcal{F}_v) \\ \mathcal{Q} = \left\{ \begin{array}{l} \texttt{frag:newFragments[}\mathcal{D}\texttt{;}\mathrm{q}_{\mathrm{item_v}}\texttt{];} \\ \mathcal{F}_1\texttt{:}\mathcal{F}_v\texttt{,frag;} \\ \mathrm{q}_{\mathrm{item}}\texttt{:(}\underline{\texttt{count}}\texttt{[}\mathcal{F}_v\texttt{]+}\underline{\texttt{key}}\ \underline{\texttt{count}}\texttt{ frag),\textbackslash:0;} \end{array} \right\}\end{array}}{\mathcal{F} \vdash \begin{array}{c} doc - lookup_{\mathsf{iter,item}} \\ | \\ \underset{\sim\sim\sim}{\triangle v} \end{array} \mapsto (\{\mathsf{iter} \rightarrow \mathrm{q}_{\mathrm{iter_v}}, \mathsf{item} \rightarrow \mathrm{q}_{\mathrm{item}}\}, \mathcal{Q}_v \parallel \mathcal{Q}, \mathcal{F}_1)} \quad (5.21)$$

Schließlich lässt sich die Übersetzung realisieren, indem `newFragments` mit den zu suchenden Dokumentennamen $\mathrm{q}_{\mathrm{item_v}}$ und der Dokumentenliste der globalen Dokumenteneinheit $\mathcal{D}$ aufgerufen wird. Die erhaltene Liste von Dokumenten wird an Fragmentliste $\mathcal{F}_v$ des Eingabe-Operatoren angehängt und bildet so die neue Fragmentliste $\mathcal{F}_1$. $\mathrm{q}_{\mathrm{item}}$ wird mit Referenzen auf das erste Element jedes erzeugten Fragments belegt. Die Fragmentnummern der Referenzen sind dabei eine Aufzählung mit `key` entsprechend der Anzahl der neuen Fragmente beginnend bei einem Offset der Anzahl der bisher bereits bestehenden Fragmente in $\mathcal{F}_v$. iter kann direkt übernommen werden.

5.16 Dokumenteneinsicht $doc - access_{\mathsf{p,n}}$

$doc - access_{\mathsf{iter,item,res}}$ bezieht für jedes Element aus item den Wert der val Spalte der derzeitig aktiven Fragmente. Die aktiven Fragmente sind als Liste von Tabellen im kdb+ Backend gespeichert (siehe Abschnitt 4.2). Die Werte in item sind hierbei Knotenreferenzen, dargestellt in kdb+ als Tupel (frag, pre). Wir definieren als Hilfsfunktion `getDocItem`, die als Eingabe eine Fragmentliste, eine Liste von Elementen sowie den Namen der Spalte der Dokumente, deren Wert zurückgegeben werden soll, erwartet. Um die schnelle Indizierung mehrerer Zeilen einer Tabelle ausnützen zu können, gruppiert die Funktion `getDocItem` vorab alle Elemente in item entsprechend ihrer Fragment-Nummer. So können alle Elemente eines Fragments auf einmal indiziert werden, was bei vielen Elementen des gleichen Fragmentes einen enormen Zeitvorteil bewirkt.

Listing 5.5: Lesen der Dokumententabelle

```
1 getDocItem:{[fragments;items];
2  fragsNo:items[;0];
3  itemsNo:items[;1];
4  groups:itemsNo[group fragsNo];
5  :raze[fragments ./: key[groups],'value[groups]];
6  };
```

In Zeile 2-3 wird `items` in seine Fragment- bzw. Knoteninformation aufgesplittet. Zeile 4 gruppiert die Fragmentinformation (vgl. Abschnitt 3.2). Das erhaltene Wörterbuch wird auf `itemsNo` als Indizes angewandt. Dadurch ergibt sich ein neues Wörterbuch `groups` mit gleichen Schlüsselwerten. Der Wertebereich enthält alle Knoten (pre) eines Fragmentes. Zeile 5 wendet schließlich `(.)` zur Indizierung in die Tiefe auf der Fragmentliste an. Zur Verdeutlichung soll erwähnt sein, dass Indizierung auf obersten Level einer Fragmentliste Tabellen spezifizieren. Indizierung auf zweiter Ebene liefert dementsprechend bestimmte Tupel dieser Tabelle. `key[groups],'value[groups]` erzeugt eine Liste von zweielementigen Tupeln der Art $(frag_i, (item_{i1}, \ldots, item_{ij}))$. Es wird also für jede Fragmentnummer und den dazugehörigen Knotenreferenzen eine Indizierung auf `fragments` vorgenommen. Durch `raze` erhält man eine flache Tabelle mit allen Knoten.

Der Übersetzung des Operators $doc - access$ besteht nun aus dem Aufruf von `getDocItem`, mit `frags` und `items` entsprechend den Werten des Kindoperators. Wie in Regel 5.22 gezeigt, wird vom zurückerhaltenen Fragment lediglich die Spalte `'val` benötigt, was das Ergebnis für $\mathrm{q_{res}}$ ergibt.

$$\frac{\mathcal{F} \vdash v \Rrightarrow (\mathcal{E}_v : \{\mathsf{item} \to \mathrm{q_{item}}, \mathsf{iter} \to \mathrm{q_{iter}}\}, \mathcal{Q}_v, \mathcal{F}_v) \qquad \mathcal{Q} = \{\ \mathrm{q_{res}}\texttt{:getDocItem[}\mathcal{F}_v\texttt{;}\mathrm{q_{item}}\texttt{][`val];}\ \}}{\mathcal{F} \vdash doc - access_{\mathsf{iter,item,res}}(v) \Rrightarrow (\mathcal{E}_v \cup \{\mathsf{res} \to \mathrm{q_{res}}\}, \mathcal{Q}_v \parallel \mathcal{Q}, \mathcal{F}_v)} \tag{5.22}$$

5.17 XPath-Step $⏍_{\alpha,n,dup}$

$⏍_{\alpha,n}$ ermittelt ausgehend von jedem Kontextknoten item für jede Partition iter alle Knoten entlang der XPath-Achse α und gibt sie als Ergebnis weiter. Die Kontextknoten item sind Referenzen der Art $(frag_{no}, item_{no})$ wie in

Kapitel 4 beschrieben. $\lrcorner_{\alpha,n,dup}$ greift auf die Fragmente in der globalen Fragmentliste zu wie sie vom Eingabe-Operator an $\lrcorner_{\alpha,n,dup}$ gegeben wird ($\mathcal{F}_v$). Der Zugriff auf die Fragmentliste wird auch hier für jedes Fragment einzeln durchgeführt wie dies auch bereits für $doc - access$ geschehen ist.

$\lrcorner_{\alpha,n,dup}$ wandelt daher die Eingabedaten in ein geeignetes Format um. Einerseits sollen die Knotenreferenzen in Fragmentinformationen sowie Knoten innerhalb der Fragmente getrennt werden. Andererseits sollen alle Eingabedaten nach Fragmenten gruppiert werden. Funktion `stepTable` (Listing 5.6) liefert eine Tabelle, die diesen Kriterien entspricht.

Listing 5.6: Tabelle für XPath-Steps

```
1 stepTable:{iter;item];
2  : 0!select iter,item by frag from
3   ([]iter:iter;item:item[;1];frag:item[;0]);
4  };
```

Wir splitten den Filter n in einen Filter auf Knotentyp *kind* und auf Knotenwert *name* auf. Wir schreiben den XPath-Step-Operator daher nun als $\lrcorner_{\alpha,kind,name}$. Regel 5.23 beschreibt die Übersetzung von $\lrcorner_{\alpha,kind,name}$.

$$\frac{\mathcal{F} \vdash v \Rightarrow (\{\ldots,\mathsf{iter} \to \mathrm{q_{iter}},\ldots,\mathsf{item} \to \mathrm{q_{item}},\ldots\},\mathcal{Q}_v,\mathcal{F}_v) \quad \mathcal{Q} = \left\{ \begin{array}{l} \texttt{tbl:stepTable[}\mathrm{q_{iter}}\texttt{;}\mathrm{q_{item}}\texttt{];} \\ \texttt{res:}\hat{\lrcorner}_\alpha\texttt{[;}\mathcal{F}_v\texttt{;}kind\texttt{;`}filter\texttt{] \underline{each} tbl;} \\ \texttt{res:\underline{value} \underline{flip} [\underline{distinct}] \underline{raze} res;} \\ \mathrm{q_{res_{iter}}}\texttt{:res[0];} \\ \mathrm{q_{res_{item}}}\texttt{:res[1],'res[2];} \end{array} \right\}}{\mathcal{F} \vdash \lrcorner^{\alpha}_{kind,name,dup,\,\mathsf{iter,item}}(v) \Rightarrow (\{\mathsf{iter} \to \mathrm{q_{res_{iter}}},\mathsf{item} \to \mathrm{q_{res_{item}}}\},\mathcal{Q}_v \parallel \mathcal{Q},\mathcal{F}_v)} \quad (5.23)$$

Gemäß der Regel werden die Eingabedaten mit Hilfe von `stepTable` geeignet zusammengestellt. Für jede Zeile dieser Formation, die jeweils allen Elementen eines Fragmentes entspricht, wird eine q Funktion aufgerufen, die durch $\hat{\lrcorner}_\alpha$[1] gekennzeichnet sei. $\hat{\lrcorner}_\alpha$ stellen dabei Hilfsfunktionen für jede XPath-Achse α dar, die den eigentlichen Lokalisierungsschritt von

[1]Man beachte, dass $\hat{\lrcorner}_\alpha$ vier Parameter besitzt. In $\mathcal{Q}$ wird der erste Parameter durch den derzeitigen Eintrag von `tbl` dargestellt. Der zweite bis vierte Funktionsparameter ist dabei konstant.

$⊿_{\alpha,kind,name}$ durchführen. Im Rahmen dieser Arbeit werden wir die Achsen `child`, `descendant` und `descendant-or-self` betrachten.
Alle $\hat{⊿}_{\alpha}$-Funktionen liefern eine Tabelle nach Schema **iter frag item**. Da $\hat{⊿}_{\alpha}$ für jeden Fragmenteintrag aufgerufen wird, liegt eine Liste von Ergebnistabellen dieses Schemas `res` vor.
Das Ergebnis des XPath-Step-Operators stellen jedoch zwei Listen iter und item dar, die Partitionsattribute sowie Knotenreferenzen beinhalten. Die Transformation von `res` zum beschrieben Ausgabeformat erfolgt in folgenden Schritten:

1. Zusammenlegen aller Tabellen der Liste `res` zu einer Tabelle mit `raze`.
2. Sollte *dup* nicht gesetzt sein, so werden alle Duplikate mit `distinct` in dieser Tabelle entfernt.
3. Invertieren der Indexreihenfolge mit `flip`. Nun liegt ein *column dictionary* vor (siehe 3.2).
4. Abstoßen des Domänenbereiches des Dictionary mit `value`. Man erhält eine dreielementige Liste mit den Inhalten der ursprünglichen Spalten.
5. Zuweisen der ersten Liste zu $\mathrm{q}_{\mathrm{res}_{\mathrm{iter}}}$ sowie die zweite und die dritte Liste zeilenweise konkateniert zu $\mathrm{q}_{\mathrm{res}_{\mathrm{item}}}$.

5.17.1 Implementierung von $\hat{⊿}_{\alpha}$

$\hat{⊿}_{\alpha}$ stellen die q Funktionen für die Auswertung der unterschiedlichen XPath-Achsen dar. In [Gru02] werden die zu erfüllenden Bedingungen der verschiedenen Achsen beschrieben. Eine Umsetzung in SQL kann durch einen Self-Join auf der Dokumententabelle und das Einfügen der Bedingungen zwischen Kontextknoten und Resultatsknoten dargestellt werden. Trotz des angebotenen `select ... from ... where`-Paradigmas, stößt man in q dabei schnell an Grenzen. Ein Self-Join in q muss unumgänglich als ein Kreuzprodukt der Dokumententabelle mit sich selbst umgesetzt werden, wobei vorhergehend eine Kopie der Dokumententabelle mit alternativen Spaltennamen angelegt werden muss. Das Kreuzprodukt im `from`-Teil der Anfrage wird zuerst voll ausgewertet und materialisiert, was durch die quadratische Speicheranforderung in Bezug auf die Dokumentengröße bereits bei sehr kleinen Dokumenten fehlschlägt.
Im folgenden werden adäquatere Vorgehensweisen für die Achsen`child`, `descendant` und `descendant-or-self` in q dargestellt. Alle $\hat{⊿}_{\alpha}$ benutzen ein Grundgerüst wie es in Listing 5.7 gezeigt ist. Die Parameter stellen dabei

den aktuellen Tabelleneintrag x mit frag, iter und item Werten, die globale Fragmentliste (fragments), den Typfilter (kindFlt) und den Knotenname (nameFlt) dar. Die Funktion wird pro Fragment aufgerufen weshalb das aktuelle Fragment aus fragment indiziert und der Hilfsvariablen tab zugewiesen wird. candidates stellen Kandidaten für die Ergebnisknoten dar. Die Erlangung dieser geschieht achsenabhängig durch die Funktion getCandidates. Die Implementierung dieser Funktion wird in folgenden Abschnitten dargelegt. candidates besitzt nach Rückgabe von getCandidates die Struktur einer Liste aus Listen, welche alle Kandidaten für jeden Kontextknoten umfassen, was bei der konkreten Implementierung deutlich wird.

Listing 5.7: Grundgerüst für $\hat{\trianglelefteq}_\alpha$

```
1  {[x;fragments;kindFlt;nameFlt];
2   tab:fragments[x['frag]];
3
4   candidates:getNodes[x['item];tab;nameFlt];
5
6   res_iter:raze[#'[(count each candidates);x 'iter]];
7   res_frag:count[res_iter]#x['frag];
8   res_item:raze[candidates];
9
10  res:flip 'iter'frag'item!
11           (res_iter;res_frag;res_item);
12  idx:where tab['kind][res_item]=kindFlt
13      and tab['name][res_item]=nameFlt;
14  :res[idx];
15  };
```

Die möglichen Ergebnisknoten müssen der Partition ihres Kontextknotens korrekt zugewiesen werden. Zeile 6 erstellt daher entsprechend der Kardinalität der Listen aller Ergebnisknoten die Partitionsliste res_iter. res ist die Ergebnistabelle nach bereits beschriebener Gestalt. Vor Rückgabe dieser werden die Filter kindFlt und nameFlt für die Ergebniskandidaten in der Dokumententabelle tab geprüft und nur die Einträge der Ergebnistabelle mit positivem Ergebnis zurückgegeben.

5.17.2 descendant-Step

Die descendant-Achse selektiert alle Knoten unterhalb des Kontextknotens. Mit Hilfe der size Informationen der Dokumentrelation lassen sich diese eindeutig bestimmen. Wir wollen also für jeden Kontextknoten *item* die Knoten bestimmen, deren Pre-Wert im Intervall $]item.\mathsf{pre}; item.\mathsf{pre} + item.\mathsf{size}]$

liegen. Wie bereits in Abschnitt 4.1 angesprochen, sind die Tabelleinträge nun aufgrund der Sortierung nach pre anhand ihrer Position in der Tabelle ansprechbar und jegliches Lesen von pre kann vermieden werden. Mit `key each` generieren wir daher die pre-Werte aller möglichen Ergebnisknoten für jeden Kontextknoten in `x[‘item]`. Für diese werden anschließend im Funktionsgerüst $\hat{\lrcorner}_{descendant}\}$ noch die Kriterien der Filter getestet. Die Funktion `getNodes` lässt sich für die `descendant`-Achse daher wie folgt darstellen:

```
1 getNodes:{[items;tab;nameFlt];
2  :(items+1)+key each tab[‘size][items];
3  };
```

`res_item` umfasst nun alle möglichen Kandidatenknoten. Für Kontextknoten $x_{item} = (x_1, \ldots, x_n)$ besitzen die möglichen Ergebnisknoten die Gestalt $candidates = ((x_1+1, \ldots, x_1+x_1.\mathsf{size}), \ldots, (x_n+1, \ldots, x_n+x_n.\mathsf{size}))$.

5.17.3 `descendant`-Step optimiert

Das Aufzählen aller möglichen Ergebnisknoten, wie es im vorherigen Abschnitt beschrieben wurde, kann einen kostspieligen Prozess darstellen, speziell bei Kontextknoten nahe der Wurzel des Dokumentes. Wird dann ein Großteil der Kandidaten durch die Filterung auf den Knotenwert wieder abgestoßen, ist diese Vorgehensweise nicht vorteilhaft. Ist ein Knotenwerttest für den Lokalisierungspfad $\lrcorner_{descendant,kind,name}$ definiert, also $name \neq \emptyset$, wird $\hat{\lrcorner}_{descendant}$ differenziert implementiert. $\hat{\lrcorner}_{descendant}$ verwendet dann sogenannte Vorgehensweise *frühe Filterung*. Dabei werden zuerst sämtliche Knoten des Dokumentes, die den Test auf *name* bestehen, als Kandidaten angesehen. Für diese wird dann schließlich geprüft ob sie im Intervall $]item; item + item.\mathsf{size}]$ der Kontextknoten liegen. Diese Intervallprüfung definieren wir in einer Hilfsfunktion `inRange`, die alle Werte des zweiten Parameters zurückgibt, die innerhalb des durch den ersten Parameter definierten Intervalls liegen. Dazu wird die q Funktion `within` benutzt. `within` liefert eine boole'sche Liste, die für jedes Element ihres ersten Parameters angibt ob er in dem durch den zweiten Paramter spezifizierten Intervall liegt. Der zweite Parameter muss eine zweielementige Liste sein, deren Elemente die Intervallsgrenzen beschreiben.

```
1 inRange:{[range;nodes];
2  :nodes[where nodes within range];
3  };
```

`getNodes` kann dann nach folgendem Muster implementiert werden:

```
getNodes:{[items;tab;nameFlt];
 grp:group[tab[‘name]][nameFlt];
 ranges:(items+1),'(items+tab[‘size][items]);
 : inRange[;grp] each ranges;
 };
```

Zeile 2 von `getNodes` erlangt nun alle möglichen Ergebnisknoten mit korrektem Knotennamen. Dies geschieht im gleichen Stile wie das bereits für den Equi-Join Operator (siehe Abschnitt 5.9.1) angewandt wurde. Es ist dabei für die Performanz von `group[‘name]` vorteilhaft, wenn das Gruppierungsattribut `‘g#` auf der Spalte name der Dokumententabelle gesetzt ist.
Bei dieser Variante von $\hat{⊿}_{descendant}$ ergibt sich für das Funktionsgerüst aus Listing 5.7 die Abänderung, dass der abschließende Namensfilter in Zeile 13 nun weggelassen werden kann.
In Abschnitt 6.3 wird der Vorteil dieser Optimierung dargestellt.
$\hat{⊿}_{\substack{descendant-\\or-self}}$ kann durch Anpassung der Intervallsbereiche nach gleichem Muster verwirklicht werden. Bei $\hat{⊿}_{\substack{descendant-\\or-self}}$ gilt für Ergebnisknoten, dass sie im Intervall $[item; item + item.\mathsf{size}]$ eines Kontextknoten $item$ liegen müssen.

5.17.4 `child`-Step

Die `child`-Achse beschränkt sich auf einen Teil der $⊿_{descendant}$-Achse. Für sie gilt die zusätzliche Einschränkung, dass bei gegebenen Kontextknoten $item$, die Ergebnisknoten $result$ folgende Bedingung erfüllen müssen: $result.\mathsf{level} = item.\mathsf{level} + 1$ Dazu definieren wir die Funktion `children`, die als ersten Paramter eine dreielementige Liste aus $(item, item.\mathsf{size}, item.\mathsf{level}+1)$ erwartet, wobei $item$ den pre-Wert eines Kontextknoten aus $⊿_{child}$ darstellt. Als zweiter Paramter wird die gesamte level-Spalte des Fragmentes übergeben. `children` zählt ähnlich wie bereits bei der `descendant`-Achse geschehen alle möglichen Ergebnisknoten auf. Anschließend prüft sie jedoch direkt, ob die betrachteten Ergebnisknoten das korrekte Level besitzen.

```
children:{[x;levelColumn]
 range:x[1]+key x[2];
 :range[where levelColumn[range]=x[0]];
 };
```

`getNodes` in 5.7 kann dann wie folgt dargestellt werden:

```
1 getNodes:{[items;tab;nameFlt];
2  :children[ ;tab[`level]] each
3    (items+1),'tab[`size][items],'tab[`level][items]+1;
4  };
```

5.17.5 child-Step optimiert

Der beschriebene Algorithmus des vorhergehenden Abschnittes ist besonders erneut sehr aufwendig bei Kontextknoten, die sehr nahe der Wurzel des Dokumentes angesiedelt sind bzw. einen sehr großen Teilbaum unter sich besitzen. Auf der Suche nach einer alternativen Lösung im Rahmen dieser Arbeit ergab sich die Vorgehensweise zum Ausnützen der parent-Information des Dokumentes. Dadurch reduziert sich getNodes auf einen Equi-Join der Kontextknoten mit der parent-Spalte der Dokumentenrelation wie in Listing 5.8 dargestellt.

Listing 5.8: getNodes als Equi-Join mit parent-Liste

```
1 getNodes:{[items;tab;nameFlt];
2  :group[tab[`parent]][items];
3  };
```

Hier ist das Gruppierungsattribut `g# auf der Spalte parent von Relevanz. Die Information aus parent kann im Rahmen dieser dieser Arbeit einzig an dieser Stelle ausgenützt werden. Jedoch erfordert sie zusätzlichen Verwaltungsaufwand bei den Konstruktoren im folgenden Abschnitt. In Abschnitt 6.4 wird ersichtlich, weshalb der erhöhte Verwaltungsaufwand bei den Konstruktoren für einen wesentlich vereinfachten Operator ⏎ in Kauf genommen wird.

5.18 Konstruktoren Λ_t

Konstrukturen bilden im Abarbeitungsplan der relationalen Algebra eine eigene Struktur, dessen Wurzelelement immer der Twig-Konstruktor Λ_{twig} darstellt. Λ_{twig} vollführt abschließende Operationen auf den Fragmenten der Eingabekonstruktoren. Die einfachsten Konstruktoren bilden $\Lambda_{textnode}$ und $\Lambda_{attribute}$. Sie generieren ein neues Fragment mit jeweils einem Knoten pro Iteration. Λ_{fcns} verschmelzt zwei Fragmente zu einem. $\Lambda_{element}$ baut ganze Elemente mit ihren Teilbäumen, was Veränderungen der Kodierungsinformationen der Knoten mit sich zieht. $\Lambda_{content}$ stellt den komplexesten Konstruktor im kdb+-Backend dar. Er konstruiert ein eigenes Fragment aus Elementen bestehender Fragmente, was diverse Anpassungen der Knoteninformation zur Folge hat.

Im kdb+-Backend generiert jeder Konstruktor ein Fragment nach dem Dokumentenschema aus Abschnitt 4. Auch Konstruktoren operieren auf verschiedenen Partitionen und diese Partitionsinformationen werden an das Ende des Schema in einer eigenen Spalte angehängt. Damit unterscheiden sich diese Fragmente in einem Konstruktorenbaum von den globalen Fragmenten der Abfrage in $\mathcal{F}$.

5.18.1 Textnode-Konstruktor $\Lambda^{textnode}_{\text{iter,item}}$

$\Lambda_{textnode}$ erstellt für jeden Partitionswert in iter einen Textknoten mit dem entsprechenden Wert aus item. Die Funktion `textnodeTemplate` liefert eine Tabelle mit einem Eintrag, der für alle Textknoten valide ist. Ein Textnode-Konstruktor besitzt keinen weiteren Konsturktor unter sich. Die zu erstellenden Knoten stellen Blätter eines XML-Fragmentes dar. Dadurch ist der Wert `0` für `size` und `level` zu setzen. Innerhalb von $\Lambda_{textnode}$ werden die Textknoten auch keinem Elternknoten zugeordnet sein, weshalb die die Referenz `parent` auf *Null* (`0N`) gesetzt wird. Die Spalte `name` ist bei Textknoten grundsätzlich leer und daher auf ‘ gesetzt[2]. Der Typ eines Textknotens ist `3`.

Listing 5.9: Muster für alle Textnodes

```
1 textnodeTemplate:{[];
2  :([] pre:    enlist 0N;
3       size:   enlist  0;
4       level:  enlist  0;
5       parent: enlist 0N;
6       kind:   enlist  3;
7       name:   enlist  `);
8  };
```

Mit der Funktion `textnodes`, die mit den Parametern q_{iter} und den Werten der zu generierende Textknoten q_{item} aufgerufen wird, wird das Ausgabefragment von $\Lambda_{textnode}$ erstellt. Sie erstellt aus dem Muster des Rückgabewertes von `textnodeTemplate` eine Tabelle der Länge der Eingabelistern. Die erstellte Tabelle wird nun zeilenweise konkateniert mit einer zweispaltigen, erstellten Tabelle mit den Parameterwerten für Knoten- und Partitionswert als Spalteninhalte. Das nun erhaltene Fragment stellt den Ergebniswert von $\Lambda_{textnode}$ dar.

[2] In q gibt es für jeden Datentypen einen eigenen *Null*-Wert. Für die Übersetzung wurde `0N` für Ganzzahlen und ‘ für Symbole benötigt.

```
1 textnodes:{[iter;val];
2  :(count[iter]#textnodeTemplate[])
3   ,' ([] val:  item;
4          iter: iter);
5  };
```

Regel 5.24 für $\Lambda_{textnode}$ besteht im wesentlichen aus dem Aufruf der Funktion `textnodes` mit den Eingabelisten als Parameter.

$$\frac{\mathcal{Q} = \{\ \mathrm{q_{frag}}\texttt{:textnodes[}\mathrm{q_{iter}}\texttt{;}\mathrm{q_{item}}\texttt{];}\ \} \quad \mathcal{F} \vdash v \Rightarrow (\{\mathsf{iter} \to \mathrm{q_{iter}}, \mathsf{item} \to \mathrm{q_{item}}\}, \mathcal{Q}_v, \mathcal{F}_v)}{\mathcal{F} \vdash \Lambda^{textnode}_{\mathsf{iter,item}}(v) \Rightarrow (\{\mathsf{frag} \to \mathrm{q_{frag}}\}, \mathcal{Q}_v \parallel \mathcal{Q}, \mathcal{F}_v)} \tag{5.24}$$

5.18.2 Attribut-Konstruktor $\Lambda^{attribute}_{\mathsf{iter,qName,item}}$

Analog zu der Vorgehensweise für $\Lambda_{textnode}$ lässt sich der Konstruktor $\Lambda_{attribute}$ übersetzen. Dafür wird eine Funktion `attributeTemplate` benötigt, die einen Tabelleneintrag nach dem Muster

pre	size	level	parent	kind
0N	0	0	0N	2

erstellt und eine Funktion `attributeNodes`, die ähnlich wie `textNodes` ausreichend Einträge für die Eingabewerte erzeugt und diese zeilenweise anhängt. Bei $\Lambda_{attribute}$ sind dies die Eingabewerte für die Tabellenspalten name, val und iter.

5.18.3 FCNS-Konstruktor Λ_{fcns}

Die Übersetzung von Λ_{fcns} ist als eine Konkatenation der beiden Eingabefragmente zu einem Ergebnisfragment anzusehen. Zusätzlich müssen wir die Referenzen der parent-Spalte der Fragmente pflegen. Die Knoten des ersten Eingabefragmentes können unangetastet bleiben, da sie ihre Position auch nach der Konkatenation behalten werden. Die parent-Werte des rechten Fragmentes müssen jedoch, um die Kardinalität der Knotenmenge des ersten Eingabefragmentes erhöht werden—jedoch nur auf ihre Partition bezogen. Wir erreichen dies durch Gruppierung nach iter des ersten Eingabefragmentes und Zählen der Elemente jeder Gruppe, wie durch die Funktion `sizeByIter` in Listing 5.10. Die nun erhaltene Tabelle nach Schema

iter	cnt

enthält für jeden iter Wert die Anzahl der Elemente im ersten Eingabefragment.

Listing 5.10: Anzahl der Elemente je Partition

```
1 sizeByIter:{[table];
2  :select cnt:count i by iter from table;
3  };
```

Funktion `fcns` aus Listing 5.11 ruft `sizeByIter` mit dem ersten Eingabefragment als Paramter auf. Durch einen Left-Join `lj` des Fragmentes von w mit dieser Tabelle erhält man für jeden Knoten des Fragmentes von w den Wert, um den parent erhöht werden muss (Zeile 5). Es gilt dabei zu beachten, dass durch den *Left Outer Join* `lj` *Null*-Werte in size entstehen können. Diese werden bevor sie auf bestehende parent-Werte addiert werden, mit `0^` auf `0` gesetzt. Schließlich wird das aktualisierte rechte Eingabefragment an das linke angehängt und zurückgegeben . Dies stellt die Ergebnisrelation von Λ_{fcns} dar.

Listing 5.11: Fragmentkonkatenation mit parent-Anpassung

```
1 fcns:{[left;right];
2  :left,
3   select pre,size,level,
4          parent:parent+0^cnt,kind,name,val,iter
5   from   right lj sizeByIter[left];
6  };
```

$$\frac{\begin{array}{c}\mathcal{F} \vdash \triangle_{v} \mapsto (\{\mathsf{frag} \rightarrow \mathrm{q}_{\mathrm{frag}_1}\}, \mathcal{Q}_v, \mathcal{F}_v) \\ \mathcal{F}_v \vdash \triangle_{w} \mapsto (\{\mathsf{frag} \rightarrow \mathrm{q}_{\mathrm{frag}_2}\}, \mathcal{Q}_w, \mathcal{F}_w) \\ \mathcal{Q} = \{\ \mathrm{q}_{\mathrm{frag}}\texttt{:fcns[}\mathrm{q}_{\mathrm{frag}_1}\texttt{;}\mathrm{q}_{\mathrm{frag}_2}\texttt{];}\ \ \}\end{array}}{\mathcal{F} \vdash \Lambda_{fcns}(\triangle_{v}, \triangle_{w}) \mapsto (\{\mathsf{frag} \rightarrow \mathrm{q}_{\mathrm{frag}}\}, \mathcal{Q}_v \parallel \mathcal{Q}_w \parallel \mathcal{Q}, \mathcal{F}_w)} \quad (5.25)$$

Besitzt Λ_{fcns} nur einen vorherhergehenden Operator bzw. ist entweder $v = \emptyset$ oder $w = \emptyset$, so lässt sich Λ_{fcns} durch eine Anwendung des Project-Operators $\pi_{\mathsf{frag:frag}}$ (siehe 5.7) auf v bzw. w darstellen.
Ist v=w=$\emptyset$, so wird eine leere Tabelle nach Schema

pre	size	level	parent	kind	name	val	iter

an frag zugewiesen.

5.18.4 Element-Konstruktor $\Lambda^{element}_{\mathsf{iter,item}}$

Der Element-Konstruktor $\Lambda_{element}$ erzeugt entsprechend der Eingabewerte für jede Partition iter ein Element mit Bezeichnung entsprechend item. Wir

benötigen hierfür erneut eine Tabelle, die ein Muster für jedes zu erzeugende Element darstellt. Dabei gehen wir von einer Funktion `elementTemplate` aus, die analog zu Abschnitt 5.18.1 ein allgemeingültiges Muster für alle Elementknoten zurückgibt. Wir erwarten für den Element-Konstruktor Einträge des Schemas

pre	size	level	parent	kind
0	0	0	0N	1

. Die gesamte Tabelle mit den Elementknoten wird nun durch die Funktion `elementTable` (Listing 5.12) erzeugt, die ähnlich zu $\Lambda_{textnode}$ das Muster eines Elementknotens vervielfältigt und die ausstehenden Spalten name, val und iter mit den Eingabewerten füllt und zeilenweise anhängt. Im Falle von $\Lambda_{element}$ bleibt die Spalte val immer leer, es muss für val jedoch *Null*-Werte für jede Zeile der neuen Tabelle erzeugt werden (Zeile 5).

Listing 5.12: Tabelle mit allen Elementknoten

```
1 elementTable:{[iter;item];
2  rows:count[iter];
3  :(rows#elementTemplate[])
4   ,' ([] name: item;
5          val:  rows#`;
6          iter: iter);
7  };
```

$\Lambda_{element}$ hängt abhängig von den Partitionswerten alle Knoten des Eingabefragmentes des rechten Kindkonstruktors als Teilbaum unter die zu erzeugenden Elemente. Daher müssen für die Elemente die endgültigen Werte der Größen der ihnen zugeordneten Teilbäume festgestellt und in size abgelegt werden. Wir gehen dabei ähnlich vor wie bereits bei Λ_{fcns} zur Bestimmung der Kardinalität der Knoten pro Partition innerhalb eines Fragmentes. Die Funktion `sizeByIter` (siehe Listing 5.10) liefert uns genau diese Werte.

Die Anpassung der size-Werte wird nun durch die Funktion `updateSizes` aus Listing 5.13 durchgeführt.

Listing 5.13: Anpassung der size-Werte

```
1 updateSizes:{[table;frag];
2  sizes:update size:cnt from sizeByIter[frag];
3  :select pre,size,level,parent,kind,name,val,iter
4   from table lj sizes;
5   };
```

Die Spalte cnt der gelieferten Tabelle von `sizeByIter` wird umbenannt in size. Anschließend wird der Left-Join-Operator `lj` mit dieser Tabelle als zweiten Parameter und der neuen Elementtabelle als ersten Parameter ausge-

führt. Die Größe der Teilbäume, die einem Element zugeordnet werden, bestimmen den Werte in size dieses Elementes. Durch diese Aktualisierung sind alle Werte der Elementknoten korrekt gesetzt.

Jedoch entstehen für die Knoten der aufgenommenen Teilbäume noch weitere Anpassungsanforderungen. Da mit den erzeugten Elementen neue Wurzelknoten der Teilbäume entstehen, ergeben sich folgende Änderungen der bestehenden Knoten im Eingabefragment:

- Inkrement der Werte in level bei bestehenden Knoten
- Die Position innerhalb des Fragmentes verschiebt sich für jeden Knoten innerhalb der Partition um eine Position. Dies hat Auswirkung auf die parent-Referenzen, die ebenso um eins erhöht werden müssen.
- Alle Knoten, deren Wert in parent bislang auf *Null* (0N) gesetzt ist, stellten Wurzelknoten in ihren Partitionen dar. Das neu erzeugte Element für diese Partition löst sie ab und so müssen die Referenzen in parent auf die neuen Elemente gesetzt werden, die die *oberste* Position einnehmen.

Diese Änderungsaktionen werden durch die in Listing 5.14 beschriebene Funktion durchgeführt.

Listing 5.14: Anpassung der level- und parent-Werte

```
1 updateLvlParent:{[frag];
2  :update level:level+1,parent:0^parent+1 from frag;
3  };
```

Listing 5.15: Element-Konstruktion

```
1 elementConstruction:{[iter;item;frag];
2  elements:elementTable[iter;item];
3  elements:updateSizes[elements];
4  :elements,updateLvlParent[frag];
5  };
```

Der Übersetzungsvorgang besteht nun durch den Aufruf der Funktion `elementConstruction` für die Eingabewerte iter, item und frag. Die Funktion koordiniert den gesamten Konstruktionsablauf beginnend mit dem Erstellen des Fragmentes mit den neuen Elementknoten und Feststellung ihrer size-Informationen. Die Knoten, die als Teilbäume der neuen Fragmente integriert werden, werden bezüglich ihrer Werte in level und parent aktualisiert bevor sie schließlich an das Fragment mit den neuen Elementen angehängt werden.

Diese erzeugte Tabelle ist der Rückgabewert von `elementConstruction` und gleichzeitig das Ergebnisfragment von $\Lambda_{element}$.

$$\frac{\begin{array}{c}\mathcal{F} \vdash v \Mapsto (\{\mathsf{iter} \to q_{\mathrm{iter}}, \mathsf{item} \to q_{\mathrm{item}}\}, \mathcal{Q}_v, \mathcal{F}_v) \\ \mathcal{F}_v \vdash w \Mapsto (\{\mathsf{frag} \to q_{\mathrm{w}}\}, \mathcal{Q}_w, \mathcal{F}_w) \\ \mathcal{Q} = \{q_{\mathrm{frag}}\texttt{:elementConstruction[}q_{\mathrm{iter}}\texttt{;}q_{\mathrm{item}}\texttt{;}q_{\mathrm{w}}\texttt{];}\quad\}\end{array}}{\mathcal{F} \vdash \underset{\mathsf{iter,item}}{\Lambda_{element}}(v, w) \Mapsto (\{\mathsf{frag} \to q_{\mathrm{frag}}\}, \mathcal{Q}_v \parallel \mathcal{Q}_w \parallel \mathcal{Q}, \mathcal{F}_w)} \tag{5.26}$$

5.18.5 Content-Konstruktor $\underset{\mathsf{iter,pos,item}}{\Lambda_{content}}$

Das Kopieren von Elementen und deren dazugehöriger gesamter Teilbaum geschieht mit dem Konstruktor $\Lambda_{content}$. Anders wie $\Lambda_{textnode}$ erstellt $\Lambda_{content}$ keine Knoten anhand der Informationen der Eingabewerten, sondern bezieht zu erstellende Knoten aus der globalen Fragmentliste $\mathcal{F}$. Wie auch bei bisherigen Konstruktoren unterscheiden wir bei der Erstellung zwischen verschiedenen Partitionen der Eingabeliste iter. Innerhalb der Partitionen können nun darüber hinaus mehrere Elemente referenziert werden durch die Knotenreferenzen in item, die entsprechend der Reihenfolge in pos angelegt werden sollen. Es wird daher auf den Knoten der Eingabemenge ein XPath-Step $\triangle_{\substack{descendant\\-or-self}}$ ausgeführt, wobei wir jedoch anstatt einer Referenz auf die pre-Werte der Einträge nun den gesamten Eintrag aus der Dokumentenrelation benötigen. Die kopierte Knotenmenge wird wie bei anderen Konstruktoren bislang auch in einer Tabelle als Ergebnisfragment zusammengefasst und zusätzlich Partitionierungsinformation in der Spalte iter angehängt. Vor Abschluss der Konstruktion müssen innerhalb dieser neu erstellten Tabelle noch die Referenzen in parent durch die neuen Position innerhalb eines Fragmentes aktualisiert werden. Gleiches gilt für die Werte in level, die auf die neue Struktur innerhalb ihrer Partitionen angepasst werden müssen.

Anordnung aller Eingabewerten

Wir benötigen eine *Arbeitstabelle* mit Partitionswert sowie Fragment- und Knotenreferenzierung. Diese muss nach Partition und Position der Elemente sortiert werden wie es in Funktion 5.16 geschieht. So wird gewährleistet, dass die Elemente des zu konstruierenden Fragmentes in korrekter Reihenfolge angeordnet sind.

Listing 5.16: Sortierte Knotentabelle

```
sortedTable:{[iter;pos;item];
 :select iter,frag:item[;0],item:item[;1]
  from `iter`pos xasc ([]iter: iter;
                         pos:  pos};
                         item: item);
 };
```

Feststellung der zu referenzierenden Teilbäume

Für jedes der Elemente benötigen wir Informationen über die Größe des dazugehörigen Teilbaumes. Wir verwenden daher die Funktion `getDocItem` wie in Listing 5.5 des $doc - access$-Operators mit den Spalten frag und item sowie dem Spaltennamen des zu beziehenden Wertes `` `size``. Wir bekommen alle size-Werte als Liste geliefert und konkatenieren zeilenweise eine einspaltige Tabelle nur mit dieser Liste an eine Tabelle, die mit `sortedTable` erzeugt wurde. Da wir zum Teilbaum der Elemente auch die Elemente selbst beziehen wollen, ist eine Erhöhung der Werte aus size um eins notwendig ($\unlhd_{\substack{descendant\\-or-self}}$).

Listing 5.17: Tabelle mit Referenzen der zu beziehenden Knoten

```
referenceTable:{[sizesTable];
 :([]
  iter:raze sizesTable[`sizes]
            #'sizesTable[`iter];
  frag:raze sizesTable[`sizes]
            #'sizesTable[`frag];
  item:raze exec item+key each sizes
            from sizesTable);
 };
```

Indizierung von Tabelleneinträgen geschieht in q durch Enumeration der benötigten Indizes. `referenceTable` mit dem Funktionsargument `sizesTable` liefert für jede Kombination von Partition und Fragmentreferenz alle Indizes der zu beziehenden Knoten des Fragmentes. Für ein bestimmtes Element erfolgt die Generierung der Indizes mit `key` entsprechend der Information aus `size`. Zu diesem erzeugten Vektor der Kardinalität des Teilbaumes wird der pre-Wert des betrachteten Elementes addiert. Führt man diese Aufzählung für jedes Element der Tabelle durch, verfügt man über alle Indizes aller Elemente und deren dazugehörigen Teilbäume.

Dokumentenzugriff

Nun greifen wir anhand der genierten Referenzen alle Einträge aus $\mathcal{F}$ ab. Wie auch bereits bei $doc - access$ führen wir pro Fragment einen Zugriff mit allen referenzierten Knoten des Fragmentes durch. Dieser Zugriff muss für jede Partition einzeln geschehen. Funktion `contentTable` in Listing 5.18 beschreibt diesen Vorgang. Sie erwartet die `references`, wie im vorherigen Schritt von der Funktion `referenceTable` erstellt, und $\mathcal{F}$ als Parameter. In Zeile 3 bis 5 des Code-Ausschnittes werden die Knotenreferenzen `item` nach Partition und Fragment gruppiert und für jeden Eintrag die Knoten – also Zeilen aus den Dokumentenrelationen in $\mathcal{F}$ bezogen. Mit `raze` erhält man nun eine einzige Tabelle mit allen bezogenen Knoten aller Partitionen und Fragmente.

Listing 5.18: Tabelle mit mit allen Knoten

```
1 contentTable:{[references;fragments];
2  entries:raze {:y[x[`frag]][x[`item]]}[;fragments]
3          each 0!select item
4                 by     iter,frag
5                 from   references;
6  entries:entries,emptyTable[];
7  :entries ,' select iter from references;
8  };
```

Zeile 6 in Listing 5.18 ist eine Absicherung, dass wir auch bei keinem zurückgelieferten Knoten im vorherigen Schritt über eine gültige *pre/size/level/parent*-Tabelle verfügen. Dazu wird eine leere Tabelle nach dem Schema

pre	size	level	parent	kind	name	val
()	()	()	()	()	()	()

durch die Funktion `emptyTable[]` erzeugt, was nicht mehr weiter erläutert wird (siehe auch 5.9). Abschließend fügt die Funktion die Partitionsinformationen an das neue Fragment an.

Aktualisierung von level und parent

Nun liegen alle benötigten Knoten in einer Fragmenttabelle vor. Diese Knoten können in diesem Zustand noch nicht weiteren Konstruktoren übergeben werden, da sich ihre Distanz zum Wurzelelement und ihre Position innerhalb eines Fragmentes verändert haben. Letzteres fordert das Neusetzen der parent-Referenzen. Dazu defnieren wir Δ_{pre} als Positionsverschiebung eines Knotens innerhalb einer Partition. Dann entspricht $\Delta_{\mathsf{pre}} = i - p$, wobei i der Index eines Eintrages in der Tabelle ist und p der pre-Wert dieses Eintrages. An dieser Stelle wird tatsächlich der pre-Wert eines Knotens betrachtet. Dies ist in der Tat die einzige Stelle an der nicht mit dem Index des Knotens in

der Tabelle gearbeitet werden kann.

Durch die Sortierung nach iter sind Knoten einer Partition immer blockweise in der derzeitigen Tabelle geführt. Da wir nur an den Verschiebungen innerhalb einer Partition interessiert sind, benötigen wir noch den Index der ersten Elemente der Parititionen als Offset für die Berechnung der neuen parent-Werte. $offset(p)$ stellt nun den minimalen Index aller Knoten einer Partition p dar.

Mit $offset(p)$ verfügen wir über alle Hilfsmittel zur Berechnung der neuen parent-Werte. Für einen Knoten n entspricht der neue parent-Wert $n.\mathsf{parent}'$ dann: $n.\mathsf{parent}' = n.\mathsf{parent} + \Delta_{\mathsf{pre}}(n) - offset(n.\mathsf{iter})$

Die Funktion `updateLvlParent` in Listing 5.19 führt die notwendigen Änderungen an den Kodierungsinformationen der Knoten durch, um diese an die Position der Knoten im neuen Fragment anzupassen. Sie erwartet das Fragment, wie es durch die bislang beschriebenen Schritte vorliegt, als Parameter. Im Detail sind folgende Änderungen durchzuführen:

- Aktualisierung von level, so dass ausgehend von den Wurzelelementen mit level $= 0$ die Werte der Teilbaumknoten angepasst werden
- Anpassung von parent an die Veränderung der Positionen Δ_{pre} der Knoten im Fragment innerhalb einer Partition
- Alle Wurzelelemente erhalten *Null*-Werte für parent

`updateLvlParent` erstellt zwei temporäre Tabellen `t0` und `t1`. `t0` beinhaltet für jeden neuen Teilbaum das bisherige minimale Level und den Index des Wurzelelementes. `t1` beinhaltet darüber hinaus $offset(n)$. In Zeile 9-14 werden die level-Werte aktualisiert, indem sie um minimale Level des Teilbaumes erniedrigt werden sowie die parent-Werte entsprechend Δ_{pre} und dem $offset(n)$ angepasst werden. Abschließend werden für alle Wurzelelemente in Zeile 16-18 die parent-Werte auf `0N` gesetzt[3].

Listing 5.19: Aktualisierung der Level- und Parent-Werte

```
1 updateLvlParent:{[fragment];
2  t0:select minLevel:min(level),roots:first i
3     by     iter,pos
4     from   fragment;
5  t1:select offset:min(roots)
6     by           iter
7     from         t0;
```

[3]q bezeichnet mit `i` in einer Tabellenabfrage den Index des Eintrages der Tabelle

```
8
9  fragment:
10     select pre,size,
11            level:level-minLevel,
12            parent:parent+(i-pre)-offset,
13            kind,name,val
14     from   fragment lj t0 lj t1;
15
16  :update parent:0N
17   from fragment
18   where i in (exec roots from t0);
19  };
```

Erstellen des Fragmentes

Die Funktion `contentConstruction` in Listing 5.20 koordiniert nun alle Schritte. Sie erstellt entsprechend ihrer Paramter `iter`, `pos` und `item` mit Hilfe der definierten Funktionen eine nach iter sortierte Tabelle der Eingabedaten und generiert alle Referenzen auf die zu beziehenden Knoten. Diese werden aus der als Parameter übergebenen Fragmentliste `frag` in ein neues Fragment kopiert und abschließend die Kodierungsinformationen bezüglich level und parent aktualisiert.

Listing 5.20: Content-Konstruktion

```
1 contentConstruction:{[iter;pos;item;frag];
2  references:referenceTable[sortedTable[iter;pos;item]];
3  newFragment:itemTable[references;frag];
4  : updateLvlParent[newFragment];
5  };
```

Regel 5.27 beschreibt nun die Übersetzung von $\Lambda_{content}$ mit Hilfe der Funktion `contentConstruction`.

$$\frac{\begin{array}{c}\mathcal{F} \vdash \triangle_v \mapsto (\{\textsf{iter} \rightarrow \mathrm{q}_{\text{iter}}, \ldots, \textsf{pos} \rightarrow \mathrm{q}_{\text{pos}}, \ldots, \textsf{item} \rightarrow \mathrm{q}_{\text{item}}\}, \mathcal{Q}_v, \mathcal{F}_v) \\ \mathcal{Q} = \{\ \mathrm{q}_{\text{frag}}\texttt{:contentConstruction[}\mathrm{q}_{\text{iter}}\texttt{;}\mathrm{q}_{\text{pos}}\texttt{;}\mathrm{q}_{\text{item}}\texttt{;}\mathcal{F}_v\texttt{];}\ \}\end{array}}{\mathcal{F} \vdash \Lambda_{content,\ \textsf{iter,pos,item}}(\triangle_v) \mapsto (\{\textsf{frag} \rightarrow \mathrm{q}_{\text{frag}}\}, \mathcal{Q}_v \parallel \mathcal{Q}, \mathcal{F}_v)} \quad (5.27)$$

Als Ergebnis fügt $\Lambda_{content}$ das neue erstellte Fragment frag in sein Environment $\mathcal{E}$ hinzu. frag wird der Rückgabewert der Funktion `contentConstruction` zugewiesen, welche entsprechend der Eingabelisten aufgerufen wird.

5.18.6 Twig-Konstruktor $\Lambda_{twig,\ \text{iter,item}}$

Die Wurzel aller Konstruktionsvorgänge stellt der Konstruktor Twig $\Lambda_{twig,\ \text{iter,item}}$ dar. Alle vorhergehenden Konstruktoren haben als Ergebnis eine Fragmententabelle, die alle Knoten aller Partitionen umfasst. Λ_{twig} ist nun zuständig für die korrekte Aufsplittung der Partitionen in eigenständige Fragmente, die schließlich als vollwertige Fragmente in der globalen Fragmentliste $\mathcal{F}$ aufgenommen werden können. Die bislang erwähnten Konstruktoren konnten eine Aktualisierung der pre-Werte der Knoten in ihren Fragmenten vernachlässigen, Die Generierung der korrekten Werte bzw. die Aktualisierung veralteter geschieht innerhalb von Λ_{twig}, um ein gültiges Fragment an die globale Fragmentliste $\mathcal{F}$ liefern zu können. Liegen alle Fragmente letztlich in passendem Format und mit korrekten Daten vor, so werden sie zu $\mathcal{F}$ hinzugefügt und Referenzen auf alle Wurzeln dieser Fragmente den Ergebnislisten von Λ_{twig} zugewiesen.

Es muss zuerst die Fragmenttabelle mit allen Knoten entsprechend ihren Partitionen gruppiert werden. Dadurch ist es möglich, für jede Partition korrekte pre-Werte für die beinhalteten Knoten zu erstellen. Dies entspricht dem Aufzählen der Werte $0 \dots n-1$ für jede Partition, wobei n die Anzahl der Knoten in der Partition ist. Die Tabelle enthält nun korrekte Werte und die Partitionen können nun in eigenständige Fragmente umgewandelt werden. Das Anwenden von `flip each`, also das Invertieren der Indizes innerhalb jeden Eintrages der Tabelle resultiert in einem Dictionary, das als Domänenbereich die Schlüsselspalte der vorherigen Tabelle (hier: iter) und im Wertebereich Tabellen enthält, die nun alle Knoten einer Partition enthalten. Funktion `twigFragList` in Listing 5.21 veranschaulicht diese Vorgehensweise.

Listing 5.21: Funktion twigFragList

```
1 twigFragList:{[fragment];
2  f0:select pre,size,level,parent,kind,name,val
3     by iter
4     from frag;
5  f0:update pre:key each count each pre from f0;
6  :flip each f0;
7  };
```

Für die Übersetzung wird die Funktion `twigFragList` mit dem Eingabefragment als Parameter aufgerufen. Der Domänenbereich entspricht der Resultatsliste `iter`. Der Wertebereich des erhaltenen Dictionary entspricht der neuen Fragmentliste, die an die globale Fragmentliste $\mathcal{F}_v$ konkateniert wird. Die Erzeugung der Referenzen auf die Wurzelelemente der neuen Fragmente geschieht wie bei $doc - lookup$ in Abschnitt 5.15.

$$
\frac{\begin{array}{c}
\mathcal{F} \vdash \triangle_{v} \Rightarrow (\{\ldots, \mathsf{frag} \rightarrow \mathrm{q}_{\mathrm{frag}}, \ldots\}, \mathcal{Q}_v, \mathcal{F}_v) \\
\mathcal{Q} = \left\{ \begin{array}{l}
\texttt{newFragments:twigFragList[}\mathrm{q}_{\mathrm{frag}}\texttt{];} \\
\mathrm{q}_{\mathrm{iter}}\texttt{:}\underline{\texttt{key}}\texttt{[newFragments][`iter];} \\
\texttt{newFragments:}\underline{\texttt{value}}\texttt{[newFragments];} \\
\mathrm{q}_{\mathrm{item}}\texttt{:(}\underline{\texttt{count}}\texttt{[}\mathcal{F}_v\texttt{]+}\underline{\texttt{key}}\ \underline{\texttt{count}}\ \texttt{newFragments),\textbackslash:0;}
\end{array} \right\}
\end{array}}{\mathcal{F} \vdash \overset{\Lambda_{twig,\ \mathsf{iter,item}}}{\underset{\triangle_v}{|}} \Rightarrow (\{\mathsf{iter} \rightarrow \mathrm{q}_{\mathrm{iter}}, \mathsf{item} \rightarrow \mathrm{q}_{\mathrm{item}}\}, \mathcal{Q}_v \parallel \mathcal{Q}, (\mathcal{F}_v; \texttt{newFragments}))} \tag{5.28}
$$

Kapitel 6

Experimente

Die Übersetzungsregeln aus dem vorherigen Abschnitt müssen noch den praktischen Test bestehen. Fokus dieser Arbeit lag auf dem *XMark*-Benchmark ([SWK+01]), der zwanzig Abfragen mit unterschiedlichen Charakteristiken umfasst. XMark umfasst zusätzlich zu den Abfragen einen XML-Dateien-Generator, *XMLgen*, anhand dessen wir für die folgenden Experimente Dateien mit den Faktoren 0,01, 0,1 sowie 1 bzw. der Größe 1,2MB, 12MB und 112 MB erzeugten.

6.1 Konfiguration

Die Testbasis für die erlangten Ergebnisse stellt eine Windows 32-Bit Workstation mit 2GHz Intel Core 2 Prozessor und 3GB RAM dar, kdb+ Version 2.4 (2007.10.03). Es existieren bereits drei verschiedene kdb+ Datenbankinstanzen für jedes XML-Dokumente der drei genannten Größen. Diese liegen vorbereitet im pre/size/level/parent-Format als *single-file* Datenbanken auf dem Hintergrundspeicher. Dazu musste vorhergehend die XML-Datei mit Hilfe des XML-Shredders, wie er schon für die Umsetzung des SQL-Backends ([May07]) benutzt wurde, in *Comma Separated Value* (CSV) - Format transformiert werden. Zum Laden der Datei sind dann folgende Schritte notwendig (siehe auch Anhang C):

1. Sortierung der Tabelle entsprechend pre
2. Attributierung der Spalten parent und name mit `g#
3. Ablegen der Tabelle auf dem Hintergrundspeicher

Die in den folgenden Abschnitten resultierenden Zeiten stellen die minimale Zeit von fünf aufeinanderfolgenden Messungen dar. Zwischen den fünf Messungen einer Abfrage ergaben sich im Laufe der Tests nur geringe Abweichungen.

6.2 Equi-Join

Bei der Übersetzung des Equi-Join Operators in Abschnitt (5.9.1) wurde eine Realisierung durch Anwendung des `group`-Operators gewählt. Das Verbinden zweier Eingaberelationen besteht daher aus einer vorbereitenden Gruppierung der Kriteriumsspalte der einen Relation mit anschließender Wörterbuchsuche für jeden Wert der Join-Spalte der zweiten Relation. Im Vergleich steht nun diese Vorgehensweise mit der alternativen, iterativen Übersetzung, die entlang des Ausdruckes `{where x=`$join_2$`} each` $join_1$ aufgebaut ist. Dazu betrachten wir die XMark Benchmark-Anfragen Q3, Q4 und Q9 auf dem 12MB-Dokument jeweils mit beiden Versionen eines Equi-Joins. Es gilt dabei zu beachten, dass sich die Wahl des Equi-Joins auch auf den Operator Theta-Join auswirkt, soweit dieser eine Gleichheitsbedingung als erstes Verbindungskriterium umfasst.

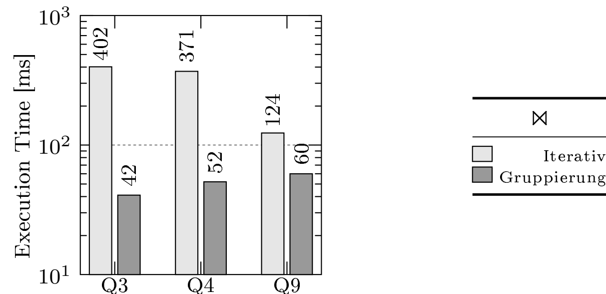


Abbildung 6.1: Equi-Join-Varianten

In Abbildung 6.1 lässt sich erkennen, wie drastisch unterlegen die iterative Variante (hellgraue Balken in der Abbildung) gegenüber dem *gruppierenden* Equi-Join ist. Die Ausführungszeiten sind in Millisekunden auf einer logarithmischen Skala dargestellt. Hier ist anzumerken, dass diese Erkenntnis auch Konsequenzen für den XPath-Step Operator hat. Erst die gruppierende Variante des Equi-Join ermöglicht ein effizientes Nutzung der parent-Information der Dokumente.

6.3 $⊿_{descendant}$: Frühzeitiges Filtern

Für die nächste Messung wird für das 12MB Dokument nun Q6,Q7,Q14 und Q19 untersucht. Sie alle beinhalten $⊿_{descendant}$-Operatoren mit Knotenwertest. In Abschnitt 5.17.2 der Übersetzung von $⊿_{descendant}$ wurden zwei Versionen der Implementierung eingeführt. Es wird nun ähnlich wie im vorherigen Abschnitt untersucht welchen Einfluss die unterschiedlichen Vorgehensweisen auf die Ausführungszeit einer vollständigen Benchmark-Abfrage besitzen.

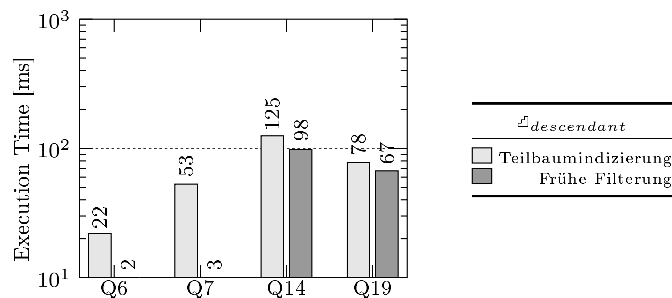


Abbildung 6.2: $⊿_{descendant}$-Varianten

Die Version *Frühe Filterung* bewirkt bei allen Abfragen eine Verbesserung der Ausführungszeit. Dabei ist eine Beschleunigung bei Q6 nicht überraschend. Q6 beinhaltet eine $⊿_{descendant}$-Operation ausgehend vom Wurzelelement des Dokumentes mit Wertfilter `site`. Das Element `site` ist eindeutig im XMark-Dokument und direkter Nachfahre des Wurzelelementes. Dieser Fall ist wie zugeschnitten auf diese Implementierungsvariante. Ähnlich in Q14 und Q19, wo ebenso ein $⊿_{descendant}$ von einem niedrigen Level im Dokument ausgeführt wird. Es lässt sich festellen, dass alle $⊿_{descendant,name,kind}$ Operationen mit Filter $name \neq \emptyset$ aus den XMark-Abfragen eine Beschleunigung erfahren haben.

6.4 $⊿_{child}$ mit parent-Werten

Eine weitreichendere Veränderung stellt die Optimierung des $⊿_{child}$-Operators dar. In 5.17.4 wurde das Ausnützen der parent-Information vorgestellt (Parent-$\bowtie$ Variante). Da die parent-Spalte der Dokumentenrelation eine eigens für diese Optimierung eingeführte Spalte darstellt und das Pflegen dieser Spalte im Rahmen der Konstruktoren einen Mehraufwand verursacht, soll nun abgewägt werden ob dieser Mehraufwand durch eine starke Beschleunigung des $⊿_{child}$-Operators mindestens aufgewogen und die Gesamtleistung verbessert werden kann. Für diese Messung werden alle XMark-Abfragen angefasst. Der $⊿_{child}$-Operator ist Bestandteil aller Abfragen und bietet somit eine große

Bandbreite an Testfällen.
Es können nun alle Abfragen eingeteilt werden in Abfragen mit bzw. ohne Konstruktoren. So lässt sich für die erste Gruppe feststellen, ob die Optimierung zumindest gegenüber ihrer Vorgänger-Implementierung eine Verbesserung zeigt und für die zweite Gruppe ob dieser Vorteil ausreicht, um die Mehrarbeit der Konstruktoren auszugleichen. Die Zeiten für das 12MB Dokument sind in Abbildung 6.3 dargestellt.

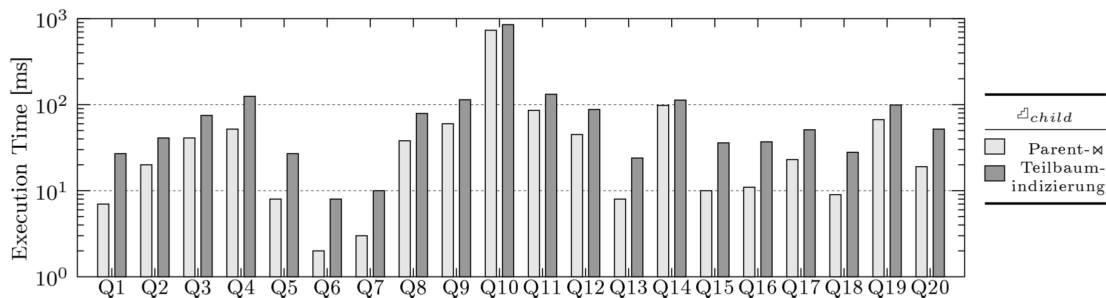


Abbildung 6.3: $\unlhd_{child}$-Varianten

Es lassen sich verschieden starke Verbesserungen durch die Anwendung der Parent-⋈ Variante auf *allen* Abfragen in Abbildung 6.3 erkennen. Abfragen mit sehr vielen Konstruktoren profitieren erwartungsgemäß nicht so stark von der Optimierung wie es Abfragen frei von Konstruktoren können. Jedoch zeigt das Ergebnis, dass sich die Aufnahme der parent-Spalte zu Gunsten eines vereinfachten $\unlhd_{child}$ für die durch die XMark-Benchmark-Abfragen abgedeckten Anwendungsfälle auszahlt.

6.5 Alle Testgrößen

Nachdem die Auswirkungen einzelner Operatorenanpassungen festgestellt wurden, werden nun alle getesteten Dokumentengrößen in Abbildung 6.4 betrachtet. Für alle Abfragen außer Q11 und Q12 ist eine lineare Skalierung eindeutig zu erkennen. Es ist nicht verwunderlich, dass gerade Q11 und Q12 schneller wachsen. Beide Abfragen besitzen das Ungleichheitsprädikat `{p/profile/@income > 5000 * exactly-one($i/text())`. Dadurch entstehen sehr große Zwischenergebnisse durch den Operator Theta-Join.

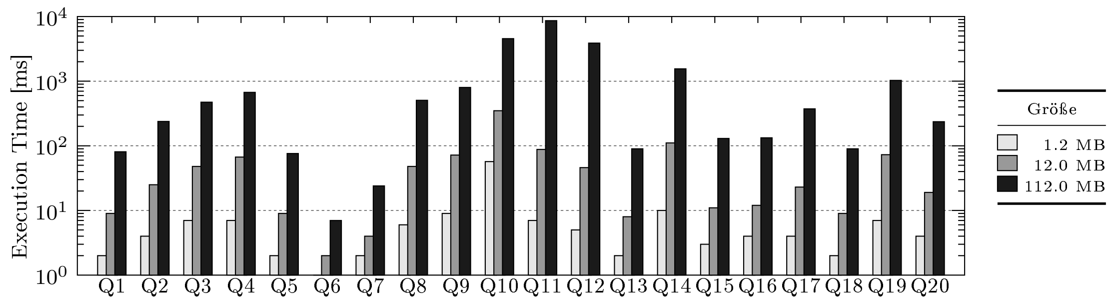


Abbildung 6.4: `XMark`-Benchmark Anfragen allen Testgröße

Abschließend lässt sich sagen, dass für eine längste Ausführungszeit von knapp unter 9 Sekunden bei Q11 die Übersetzung zu kdb+ sich durchaus erfolgreich präsentiert. Ein Großteil der Abfragen lässt sich weit unter einer Sekunde für das 112MB Dokument bzw. unter 100 Millisekunden für das 12MB Dokument durchführen.

Kapitel 7

Resümee

Ziel dieser Diplomarbeit ist die Übersetzung der relationalen Algebra von Pathfinder zu kdb+. Ein großer Anteil der Operatoren der Pathfinder Algebra ist durch diese Arbeit dabei abgedeckt. Die XMark-Benchmark-Abfragen werden vollständig ausgeführt wie in den Experimenten gesehen und produzieren korrekte Ergebnisse.

Im Laufe der Entwicklung der Übersetzungsregeln wurde die Performanz der erstellten q Skripte extrem gesteigert. Betrachtet man die aktuellen Ergebnisse der Ausführungszeiten von Q10 auf einem 112MB Dokument, so hat sich diese enorm verbessert. Ein erster Lauf benötigte weit über 30 Minuten und konnte auf letztlich ca. fünf Sekunden reduziert werden. Eine inkrementelle Verbesserung der aufwändigen Konstruktoren hatte daran ihren Anteil. Stück für Stück flossen die Erkenntnisse aus anderen Operatoren dort ein. Das Vermeiden von `each`-Iterationen, das Kleinhalten von Zwischenergebnissen und die Anwendung des *gruppierenden* Equi-Joins machten teilweise enorme Unterschiede für die Ausführungszeit einzelner Operationen aus.

Ein weiterer wichtiger Bestandteil der Verbesserung der Ausführungszeiten war die Einführung der `parent`-Spalten, die es ermöglichten, den XPath-Step für die Achse `child` als einen Equi-Join auf der Dokumentenrelation und den Kontextknoten darzustellen. Darüber hinaus kann in Aussicht gestellt werden, dass die `parent`-Information sich auch für weitere XPath-Achsen als hilfreich erweisen werden.

Die sehr schnelle Ausführungszeiten der XMark-Benchmarks in kdb+ können erreicht werden durch das exzessive Ausnutzen von *positional lookups*. Durch die vorhergehende Sortierung des relationalen Layouts der Daten nach den Pre-Order Werten, können alle Zugriffe auf das Dokument einschließlich

der XPath-Lokalisierungsschritte extrem beschleunigt werden. *Positional lookups* sind weit schneller als jegliche Vorgehensweise von wertbasiertem Aufsuchen einschließlich der in kdb+ angebotenen, Index-ähnlichen Attributierung von Listen mit `` `g# ``.

Diese Arbeit hat gezeigt, dass kdb+ für XQuery eine hervorragende Plattform darstellt. Alle Übersetzungen konnten in der systemeigenen Programmiersprache q realisiert werden. Praktisch über alle Operatoren hinweg wurden q-typische Paradigmen verwendet und untypische wie `do`, `while` oder ähnliche Schleifen sowie Verzweigungen wie `if` konnten komplett außen vor gelassen werden.
Schließlich lässt sich feststellen, dass Pathfinder und kdb+ eine hervorragende Kombination bilden.

Anhang A

Der Prototyp `pf2kdb`

Die Übersetzungsregeln von der relationalen Algebra Pathfinders zu kdb+ aus Kapitel 5 können bislang von einem Java-Prototypen zur Anwendung benutzt werden. Dieser basiert auf Pathfinder (Stand: 10/09/2007) Algebra XML Output. Zur Durchführung einer XQuery-Abfrage auf einem kdb+-Backend sind also folgende Schritte notwendig:

```
$ pf -AlX -s17 -fSLK q01.xq > q01_xq.xml
$ java -jar <path-to-pf2kdb-workdir>/pf2kdb.jar \
       -in q01_xq.xml -q
```

Die Option `-q` bewirkt eine direkte Ausführung des erzeugten q Codes auf der kdb+-Instanz `localhost:5001`, was ggf. abgeändert anzugeben ist. Voraussetzung dafür ist eine laufende kdb+ - Instanz, die davor mit folgendem Befehl gestartet wurde und auf Port `5001` ansprechbar ist.

```
$ q <path-to-database>/ -p 5001
```

Anhang B

Beispielübersetzung von Q2

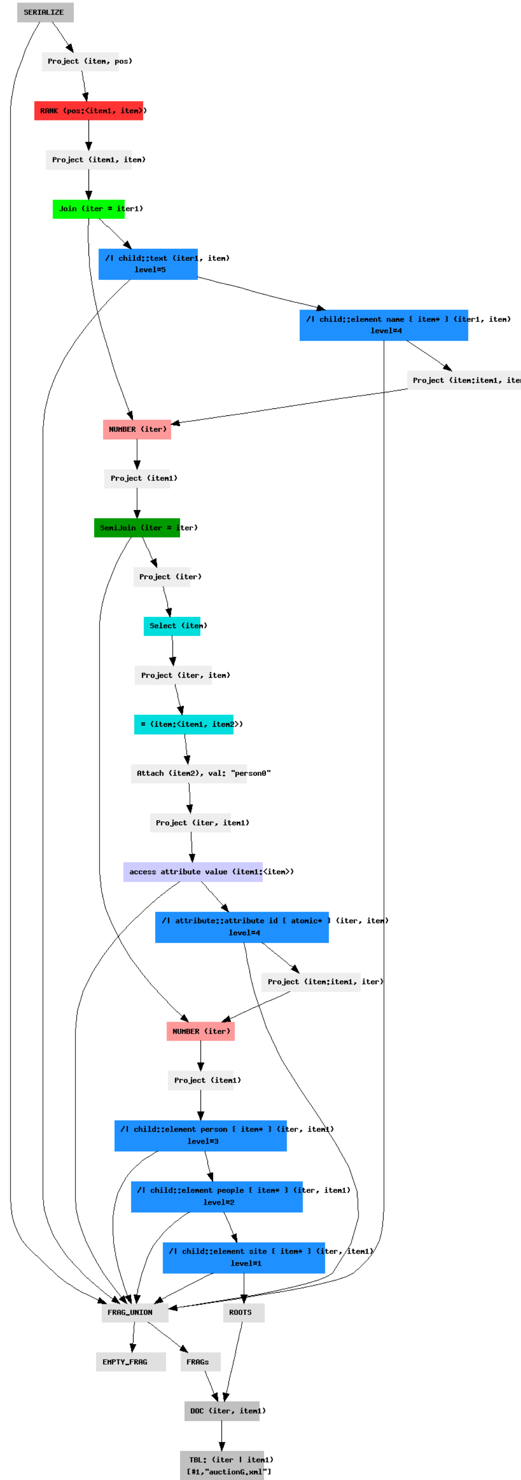
B.1 Abfrage in XQuery

```
let $auction := doc("auctionG.xml") return
for $b in $auction/site/people/person[@id = "person0"]
return $b/name/text()
```

B.2 Serialisiertes XML Ergebnis

```
$ java -jar <path-to-pf2kdb-workdir>/pf2kdb.jar \
  -in q01.xq.xml -q -s
<?xml version="1.0" encoding="utf-8"?>
<XQueryResult>Ayonca Vijaykrishnan</XQueryResult>
```

B.3 Abfrage als Pathfinder Algebra Plan



B.4 Generierter q Code

```
1 fragList:();
2 nxt_frag:0;
3
4 childStep:{[x;fragments;kindFlt];
5  tab:fragments[x `frag];
6  res_item:group[tab `parent][x `item];
7  res_iter:raze[#'[(count each res_item);x `iter]];
8  res_frag:count[res_iter]#x `frag;
9  res_item:raze res_item;
10  idx:where tab[`kind][res_item]=kindFlt;
11  :(flip `iter`frag`item!(res_iter;res_frag;res_item))[idx];
12  };
13
14 childStepFilter:{[x;fragments;kindFlt;nameFlt];
15  tab:fragments[x `frag];
16  res_item:group[tab `parent][x `item];
17  res_iter:raze[#'[(count each res_item);x `iter]];
18  res_frag:count[res_iter]#x `frag;
19  res_item:raze res_item;
20  idx:where (tab[`kind][res_item]=kindFlt) and tab[`name][res_item]=nameFlt;
21  :(flip `iter`frag`item!(res_iter;res_frag;res_item))[idx];
22  };
23
24 / table, id 6
25 a0:raze[enlist[1]];
26 a1:raze[enlist[`$"auctionG.xml"]];
27
28 / fn:doc, id 5
29 a3:(`$".pf.doc." ,/:raze string {tables[`$".pf.doc"]
30     [where x=.\:[value each `$".pf.doc.",/:string tables `$".pf.doc";(0; `val)]]}
31     each a1);
32 fragList:fragList,({select from value x} each a3);
33 a3:+[nxt_frag;key[count[a3]]];
34 +:[nxt_frag;count[a3]];
35 a3:,\:[a3;0];
36
37 / XPath step, id 18
38 t1:([]iter:a0;item:a3[;1];frag:a3[;0]);
39 t1:0! select iter,item by frag from t1;
40 t2:value flip raze[(childStepFilter[;fragList;1;`$"site"] each t1)];
41 a2:t2[0];
42 a4:,'[t2[1];t2[2]];
43
44 / XPath step, id 17
45 t1:([]iter:a2;item:a4[;1];frag:a4[;0]);
46 t1:0! select iter,item by frag from t1;
47 t2:value flip raze[(childStepFilter[;fragList;1;`$"people"] each t1)];
48 a5:t2[0];
49 a6:,'[t2[1];t2[2]];
50
51 / XPath step, id 16
52 t1:([]iter:a5;item:a6[;1];frag:a6[;0]);
53 t1:0! select iter,item by frag from t1;
54 t2:value flip distinct raze[(childStepFilter[;fragList;1;`$"person"] each t1)];
55 a7:t2[0];
56 a8:,'[t2[1];t2[2]];
57
58 / number, id 14
59 a9:1+key count[a8];
```

```
60
61 / XPath step, id 27
62 t1:([]iter:a9;item:a8[;1];frag:a8[;0]);
63 t1:0! select iter,item by frag from t1;
64 t2:value flip raze[(childStepFilter[;fragList;2;`$"id"] each t1)];
65 a10:t2[0];
66 a11:,'[t2[1];t2[2]];
67
68 / #pf:string-value, id 26
69 t1:a11[;1][group[a11[;0]]];
70 a12:raze[./:[fragList;,'[key[t1];enlist value[t1]]]][`$"val"];
71
72 / attach, id 24
73 a13:count[a12]#`$"person0";
74
75 / eq, id 23
76 a14:raze[enlist[=[a12;a13]]];
77
78 / select, id 21
79 t1:where[a14];
80 a15:a10[t1];
81 a16:a14[t1];
82
83 / semijoin, id 13
84 t1:where a9 in a15;
85 a17:a8[t1];
86 a18:a9[t1];
87
88 / number, id 11
89 a19:1+key count[a17];
90
91 / XPath step, id 30
92 t1:([]iter:a19;item:a17[;1];frag:a17[;0]);
93 t1:0! select iter,item by frag from t1;
94 t2:value flip raze[(childStepFilter[;fragList;1;`$"name"] each t1)];
95 a20:t2[0];
96 a21:,'[t2[1];t2[2]];
97
98 / XPath step, id 29
99 t1:([]iter:a20;item:a21[;1];frag:a21[;0]);
100 t1:0! select iter,item by frag from t1;
101 t2:value flip distinct raze[(childStep[;fragList;3] each t1)];
102 a22:t2[0];
103 a23:,'[t2[1];t2[2]];
104
105 / eqjoin, id 10
106 t3:group[a22][a19];
107 t4:raze #'[(count each t3);key[count[t3]]];
108 t5:raze[t3];
109 a24:a17[t4];
110 a25:a19[t4];
111 a26:a22[t5];
112 a27:a23[t5];
113
114 / rank, id 8
115 a28:rank[(a24,'a27)];
116
117 / serialize, id 1
118 `pos xasc ([]pos:a28;item:a27)
```

Anhang C

Laden von CSV Dateien

Ein Vorschlag für das Einlesen einer pre/size/level/parent CSV-Datei mit persistentem Speichern in einer kdb+ Datenbank.

```
reloadDB:{[dbDir];
 sym::get `$string[dbDir],"/sym";
 value "\\l ",1_ string[dbDir];
 };

loadCSV:{[tableName;csvFile;dbDir;docNs;splayed];
 docPath:string[dbDir],"/",string[docNs];
 maxDoc:count@[tables;docNs;()];
 newTab:flip `pre`size`level`parent`kind`name`val!
        ("IIIIISS";",") 0: csvFile;
 newTab:`pre xasc select from newTab;
 newTab:update `g#kind,`g#parent,`g#name from newTab;
 newTab[0;`val]:tableName;
 if[splayed;
   $[`;docPath,".t",string[maxDoc],"/"]
              set .Q.en[dbDir] newTab;
   reloadDB[dbDir];
 ];
 if[not splayed;
  $[`;docPath,".t",string[maxDoc]] set newTab;
  value "\\l ",1_ string[dbDir];
 ];
 };
```

Beispielhafte Verwendung:

```
q)\l loadCSV.q
q)loadCSV[`auctionG.xml;`:/kx/data/xmark_gen_f1.csv;`:/kx/db_parent;`.pf.doc;0b];
```

Anhang D

Beigefügte DVD

`data/xml/`	XML-Dokumente der Experimente, generiert mit *XMLgen*
`data/csv/`	Kodierte XML-Dokumente als CSV
`data/kdb/`	kdb+ Datenformat der einzelnen Dokumente
`query/xq/`	Original XMark Abfragen
`query/alg/`	XMark Abfragen in XML Format der relationalen Algebra
`query/gif/`	Algebra Pläne als `gif`-Bilddateien
`src/`	Der Java Quellcode des Prototypen `pf2kdb`
`dist/`	`pf2kdb` kompiliert ausführbar
`diplomarbeit.pdf`	Dieser Text im PDF-Format

Literaturverzeichnis

[BGvK+05] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Loop-Lifted Staircase Join: From XPath to XQuery. Technical report, CWI, Amsterdam, The Netherlands, March 2005.

[Bru04] M. Brundage. *XQuery: The XML Query Language.* Addison Wesley, 2004.

[CD99] J. Clark and S. DeRose. XML Path Language (XPath), 1999. http://www.w3.org/TR/xpath.html.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[EN04] R. Elmasri and S. Navathe. *Fundamentals of Database Systems.* Addison Wesley, 2004.

[Gru02] T. Grust. Accelerating XPath Location Steps. *In Proceedings of the 21 International ACM SIGMOD Conference on Management of Data, Madison, Wisconsin, USA*, pages 109–120, June 2002.

[GST04] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. Technical report, Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004.

[KE06] A. Kemper and A. Eickler. *Datenbanksysteme - Eine Einführung.* Oldenbourg Wissenschaftsverlag GMBH, 2006.

[kx] kdb+. www.kx.com.

[May07] M. Mayr. Ein SQL:99 Codegenerator für Pathfinder. Master's thesis, Technische Universität München, 2007.

[O'N06] Michael O'Neill. KDB+ Reference Manual 3.0, 2006. www.firstderivatives.com.

[Ort06] Don Orth. Q Language Reference Manual, 2006. http://kx.com/q/d/q1.htm.

[pat] Pathfinder. www.pathfinder-xquery.org.

[Sha05] Dennis Shasha. Kdb+ Database and Language Primer, 2005. http://kx.com/q/d/primer.htm, shasha@cs.nyu.edu.

[Ste01] A. Steger. *Diskrete Strukturen Band 1*. Springer, 2001.

[SWK+01] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, June 2001.

[Sys06] Kx Systems. Kdb+ Database white paper, 2006. info@kx.com.

[Teu06] J. Teubner. Pathfinder: XQuery Compilation Techniques for Relational Database Targets, 2006.

[W3C07] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, 2007.

Abbildungsverzeichnis

1.1 XQuery auf kdb+ mit Pathfinder 2

2.1 Konstruktorenbaum 12

6.1 Equi-Join-Varianten 67
6.2 $⊿_{descendant}$-Varianten 68
6.3 $⊿_{child}$-Varianten 69
6.4 `XMark`-Benchmark Anfragen allen Testgröße 70

Tabellenverzeichnis

2.1 Operatoren der relationalen Algebra 5
2.2 Addition als Beispiel für Tupelfunktion 6
2.3 Anwendung der Tupelnummerierung $\rho_{\langle a_1,a_2\rangle,b,p}$ 8

4.1 XPath-Achsenprädikate 24

Index

bag und *set* - Semantik, 7

Beispielübersetzung, 73

count, 16

Dictionary, 18

enlist, 15
Experimente, 66

FCNS - First Child Next Sibling, 11
Fragmente, 25

group, 18

Joins, 21

kdb+/q, 13
key, 18
Konfiguration, 66
Konkatenation, 16
Konstruktorenbaum, 12

Listen, 15

Operatoren-Übersetzung, 28
- Aggregation, 34
- attach, 31
- Attribut-Konstruktion, 55
- Boole'sche Vergleiche, 32
- Content-Konstruktion, 59
- Distinct, 40
- Dokumenteneinsicht, 46
- Dokumentensuche, 45
- Duplikateliminierung, 40
- Element-Konstruktion, 56
- Equi-join, 35
- FCNS-Konstruktion, 55
- Folgerungsregel, 29
- Konstruktoren, 53
- Kreuzprodukt, 42
- Mengenoperationen, 41
- Projektion, 34
- Selektion, 33
- Semi-Join, 37
- Tabelle, 30
- Textnode-Konstruktion, 54
- Theta-Join, 38
- Tupelfunktionen, 32
- Tupelnummerierung, 43
- Twig-Konstruktion, 64
- Typumwandlung, 44
- XPath-Step, 47

Positional Look-Ups, 14

q, 14

Relationale Algebra, 4
- *Path-Step*-Operator, 9
- Aggregation, 6
- Attach, 5
- Boole'sche Vergleiche, 5
- Dokumenteneinsicht, 9
- Dokumentensuche, 9
- Duplikateliminierung, 7
- Joins, 7
- Konstruktoren, 10
- Kreuzprodukt, 8
- Mengenoperationen, 8

Projektion, 6
Selektion, 6
Tabelle, 4
Tupelfunktionen, 5
Tupelnummerierung, 8
Typumwandlung, 9
Relationales Layout, 22

Selektivität, 20
Serialisierung, 28
Spaltenorientierung, 13
Subtree Copy, 11
Symbol, 15

value, 18
Verbundoperationen, 21

Wörterbuch, 18

XML-Repräsentation, 22
XPath, 9

Zuweisung, 14