Hochschule für Technik, Wirtschaft und Kultur (HTWK)

Fakultät Informatik, Mathematik und Naturwissenschaften

Masterstudiengang Informatik

Masterarbeit

zur Erlangung der akademischen Grades

Master of Science (M.Sc.)

Implementierung einer NoSQL-Query-Language

von Max Taube

Leipzig, 27. Dezember 2016

Betreuer: Prof. Dr.-Ing. Kudraß

Zweitgutachter: Prof. Dr. rer. nat. Thomas Riechert

Abstrakt

Document-Stores sind NoSQL-Datenbanken und halten semi-strukturierte Daten, oft modeliert als JSON. Die verfügbaren Abfragemöglichkeiten sind limitiert. Komplexe Query-Languages, wie SQL für relationale Datenbanken, sind durch das fehlende Schema nur erschwert umsetzbar. Dennoch haben einige NoSQL-Datenbanken, wie Couchbase mit N1QL oder AsterixDB mit AQL, umfangreiche Query-Languages implementiert. Mit SQL++ wurde versucht eine vereinheitlichende Sprache zu definieren, welche semantisch direkt auf andere semi-strukturierte Query-Languages übertragbar ist.

Das Ziel dieser Arbeit ist eine solche Sprache in der Datenbank "Cheesebase" zu implementieren. "Cheesebase" ist eine selbst programmierte Datenbank, welche als JSON modelierbare Daten speichert und durch minimale Methoden zur Verfügung stellt. Im laufe der Arbeit wird eine an SQL++ orientierte Query-Language definiert und implementiert. Um eine effiziente Ausführung der Queries zu erlauben wird der interne Datenzugriff der Datenbank angepasst und erweitert.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1

1.2 Cheesebase 2

1.3 Existierende Sprachen 2

1.3.1 MongoDB Query Language 2

1.3.2 AsterixDB Query Language 3

1.3.3 JSONiq 3

1.3.4 N1QL 3

2 SQL++ **4**

2.1 Datenmodell 4

2.1.1 Vergleich mit SQL 6

2.2 Queries 6

2.2.1 Pfade 6

2.2.2 Variablenbindungen 7

2.2.3 SFW-Queries 8

2.2.4 SELECT-Klausel 10

2.2.5 FROM-Klausel 11

2.2.6 Andere Klauseln 14

2.2.7 Konfiguration 16

2.2.8 Algebra und Funktionen 16

3 Language-Processor **17**

3.1 Interpreter-Strategie 18

4 Abstract-Syntax-Tree **20**

4.1 Vererbung gegen Vereinigung 20

4.2 Expression 21

4.3 Variablen 22

4.4 Komplexe Werte 22

4.5 Pfad-Navigation . . . 23
4.6 Funktionen . . . 23
4.7 Operatoren . . . 24
4.8 SFW-Query . . . 24
4.8.1 SELECT . . . 25
4.8.2 FROM . . . 25
4.8.3 WHERE und HAVING . . . 26
4.8.4 GROUP BY . . . 26
4.8.5 ORDER BY . . . 27
4.8.6 LIMIT und OFFSET . . . 27

5 Datentyp **28**

6 Parser **30**
6.1 Grundlagen . . . 31
6.1.1 Grammatik . . . 31
6.1.2 Lexikalische- und Syntaktische-Analyse . . . 32
6.2 Boost Spirit X3 . . . 33
6.2.1 Minimaler Parser . . . 33
6.2.2 Zeichenketten Parser . . . 34
6.2.3 Kombinierter Parser . . . 34
6.2.4 Regeldefinition . . . 35
6.2.5 Alternativen . . . 36
6.2.6 Semantic-Actions . . . 36
6.2.7 Zusammenfassung . . . 37
6.3 Expression . . . 37
6.4 Werte . . . 38
6.5 Variablen . . . 39
6.6 Funktionen . . . 39
6.7 Komplexe Werte . . . 39
6.8 Pfad-Navigationen . . . 40
6.9 Operatoren . . . 43
6.10 SFW-Query . . . 44
6.10.1 SELECT . . . 46
6.10.2 FROM . . . 47
6.10.3 WHERE . . . 49
6.10.4 ORDER BY . . . 49
6.10.5 GROUP BY . . . 50
6.10.6 LIMIT und OFFSET . . . 50

6.11 Beispiele 50

7 Ausführung **53**
7.1 Umgebungen 54
7.2 Expression 55
7.3 Werte 56
7.4 Variable 56
7.5 Komplexe Werte 57
7.6 Navigationen 57
7.7 Operatoren 58
7.8 SFW-Query 58
7.8.1 FROM 59
7.8.2 WHERE 65
7.8.3 GROUP BY 66
7.8.4 HAVING 67
7.8.5 ORDER BY 67
7.8.6 LIMIT und OFFSET 68
7.8.7 SELECT 68
7.9 Funktionen 69

8 Datenanbindung **72**
8.1 Deadlocks 72
8.2 Einfache Implementierung 72
8.3 Beispiel 73
8.4 Lazy Implementierung 74

9 Mögliche Optimierungen **76**
9.1 Statische Teilbäume 76
9.2 Indizes 77

10 Fazit **78**
10.1 Zusammenfassung 78
10.2 Beispiele 78
10.3 Einschätzung von SQL++ 79

Abbildungsverzeichnis

2.1 SQL Tabelle als SQL++-Wert 6
2.2 SFW-Query Auswertung 9

3.1 Language-Processor Typen 17

4.1 Vererbung gegen Union 21
4.2 Speicherdarstellung des AST einer Pfad-Navigation 23

6.1 Query-Parsing 30
6.2 Ableitungsbaum des Beispiels 32
6.3 TupleNav-Parser 42
6.4 Beispiel Parsetree 45
6.5 Beispiel AST 45

7.1 Query-Execution 53
7.2 Struktur einer Umgebung 55

8.1 Varianten eines Tupel 75

Darstellungsverzeichnis

1.1 MongoDB Query Beispiel 2
1.2 AQL Beispiel 3
1.3 JSONiq Beispiel 3
2.1 SQL++ Daten-Grammatik 5
2.2 SQL++ Query-Grammatik 7
2.3 SELECT-Klausel 10
2.4 FROM-Klausel 11
4.1 AST: Expression 21
4.2 AST: Variable 22
4.3 AST: komplexe Werte 23
4.4 AST: Pfad-Navigation 23
4.5 AST: Funktion 24
4.6 AST: Operatoren 24
4.7 AST: SFW-Query 24
4.8 AST: SELECT-Klausel 25
4.9 AST: FROM-Klausel 25
4.10 AST: WHERE/HAVING-Klausel 26
4.11 AST: GROUP BY-Klausel 27
4.12 AST: ORDER BY-Klausel 27
4.13 AST: LIMIT/OFFSET-Klausel 27
5.1 Datenmodell Implementierung 28
6.1 Spirit: Integer Parser 33
6.2 Spirit: String Parser 34
6.3 Spirit: kombinierter String Parser 34
6.4 Spirit: Regeldefinition 35
6.5 Spirit: Alternativen 36
6.6 Spirit: Semantic-Actions 37
6.7 Parser: Expression 37
6.8 Parser: Value 38
6.9 Parser: Variable 39

6.10 Parser: Funktion . . . 39
6.11 Parser: Sammlungen . . . 40
6.12 Parser: naive Tupel-Navigation . . . 40
6.13 Parser: angepasste Tupel-Navigation . . . 41
6.14 Parser: finale Tupel-Navigation . . . 43
6.15 Parser: Operatoren . . . 44
6.16 Parser: Expression-Ebenen . . . 46
6.17 Parser: SFW-Query . . . 46
6.18 Parser: SELECT-Klausel . . . 46
6.19 Parser: SELECT-Liste . . . 47
6.20 Parser: FROM-Klausel . . . 47
6.21 Parser: Basis FROM-Items . . . 48
6.22 Parser: INNER|LEFT|FULL CORRELATE . . . 48
6.23 Parser: INNER JOIN . . . 49
6.24 Parser: WHERE . . . 49
6.25 Parser: ORDER BY . . . 49
6.26 Parser: GROUP BY . . . 50
6.27 Parser: LIMIT/OFFSET . . . 50
7.1 Implementierung von Umgebung . . . 54
7.2 Evaluierung einer Expression . . . 56
7.3 Evaluierung eines Wertes . . . 56
7.4 Evaluierung von komplexen Werten . . . 57
7.5 Implementierung des Plus-Operators . . . 58
7.6 Implementierung der SFW-Query Evaluation . . . 59
7.7 Implementierung der Evaluation von FromEmpty . . . 60
7.8 Pseudocode der Evaluierung von FromTuple . . . 61
7.9 Pseudocode der Evaluierung von FromInner . . . 62
7.10 Pseudocode der Evaluierung von FromLeft . . . 63
7.11 Pseudocode der Evaluierung von FromFull . . . 65
7.12 Pseudocode der Evaluierung von GroupBy . . . 67
7.13 Pseudocode der Evaluierung von Funktionen . . . 71
8.1 Query-Funktion in Cheesebase . . . 73

Kapitel 1

Einleitung

1.1 Motivation

Relationale Datenbanken sind weit verbreitet und das dabei verwendete relationale Datenmodell ist mächtig und in der Lage fast alle Strukturen von Daten darzustellen. SQL ist ein allgemein akzeptierter Standard für Anfragen an relationale Datenbanken. Jedoch ist das relationale Datenmodell nicht immer optimal. Viele Anwender wünschen eine Vielseitigkeit der Datenbank, die Möglichkeit die Struktur der Daten in bestimmten Bereichen zu ändern, ohne das gesamte Schema anzupassen. Auch das Darstellen von verschachtelten Daten, ohne die Definition einer Vielzahl von Tabellen, spricht für ein nicht-relationales Datenmodell. Durch den aktuell großen Fokus auf verteilte Systeme sind Dokument-orientierte Datenbanken sehr populär geworden.

Durch die hierarchische Struktur der Dokument-orientierten Daten können viele Anfragen durch das Lesen einzelner Werte erfüllt werden. Eine umfangreiche Query-Language wird in vielen Anwendungsfällen dennoch benötigt. So kann eine Transformation, Filterung oder Kombination von Daten in einer Anfrage gewünscht sein.

Für einzelne Dokumente existieren Sprachen wie XPath oder XQuery, welche jedoch nicht praktisch in Datenbanksystemen nutzbar sind. Die Entwickler von Dokument-orientierten DBMS stellen oft direkt für die Anwendung entwickelte Query-Languages bereit. Diese sind sehr verschieden und orientieren sich meist an den Funktionen des betroffenen DBMS. Es existieren auch allgemeine Dokument-orientierte Query-Languages, jedoch hat keine eine weitläufige Anwendung, wie SQL, gefunden.

Die verschachtelte und heterogene Struktur der Dokument-orientierten Daten ist der Hauptunterschied zu relationalen Daten. In einer Query kann der Typ eines Datums erst beim lesen dieses erkannt werden. Durch die Verschachtelung kann jedes Datum

selbst aus einer Struktur weiterer Daten bestehen, somit ist es nötig dass eine Query-Language das Resultat einer Operation weiter verarbeiten und navigieren kann. Weitere Aspekte die eine Dokument-orientierte Query-Language unterstützen muss sind die tiefe Navigation in Hierarchien und korrekte Verarbeitung von strukturell nicht existierenden Daten.

1.2 Cheesebase

Cheesebase [10] ist ein DBMS im Format einer Programmbibliothek, geschrieben in C++. Sie hält als JSON modellierbare Daten und speichert diese in einer einzelnen Datei. Der Inhalt der Datenbank ist als einziges großes JSON-Dokument darstellbar. Cheesebase erlaubt das Einfügen, Abfragen und Ändern von Werten an beliebigen Positionen. Die Bibliothek ist alleinstehend, kompakt, und benötigt keine Konfiguration oder Installation. Somit kann sie einfach in andere Projekte eingebunden und verwendet werden.

1.3 Existierende Sprachen

1.3.1 MongoDB Query Language

MongoDB akzeptiert Queries in Form einer Methode. Die `find`-Methode wird auf einem Dokument ausgeführt und akzeptiert einen Filter sowie eine Projektion als Argumente. Von dieser wird eine Daten-Sequenz zurückgegeben welche durch weitere Methoden, wie `sort`, modifiziert werden kann. Aggregatfunktionen werden durch eine `aggregate`-Methode direkt auf einem Dokument durchgeführt. Eine Beispiel-Query ist in Darstellung 1.1 zu sehen. [3]

Die Query-API ist stark an der Architektur von MongoDB orientiert und bezieht sich somit auf einzelne Dokumente. Queries welche Daten aus verschiedenen Dokumenten kombinieren sind nicht direkt unterstützt.

Darstellung 1.1 MongoDB Query Beispiel

```
db.people.find(
  { age: { $gt: 30 } },
  { name: 1 }
)
```

1.3.2 AsterixDB Query Language

AQL orientiert sich an XQuery und gleicht einer funktionalen Programmiersprache. Das heißt dass Ausdrücke als Funktionsaufrufe zu verstehen sind, die einen Rückgabewert liefern der als Argument einer weiteren Funktion genutzt werden kann. Das zentrale Element stellt die FLWOR (FOR, LET, WHERE, ORDER BY, RETURN) Expression dar. In Darstellung 1.2 ist eine Beispiel-Query dargestellt.

Darstellung 1.2 AQL Beispiel

```
for $person in dataset People
where $person.age > 30
return { "name": $person.name }
```

1.3.3 JSONiq

JSONiq ist nahezu identisch zu AQL. JSONiq dient jedoch nicht als Query-Language eines bestimmten DBMS, sondern als Sprache für JSON-Dokumente aus verschiedenen Quellen. So können Daten aus Dateien, online Quellen oder, durch das nutzten von Plugins, verschiedenen Datenbanken abgerufen und manipuliert werden. JSONiq wirbt als Middleware und soll verschiedene Datenbanken als Datenquellen einbinden können um eine einheitliche Query-Language bereit zu stellen.

1.3.4 N1QL

N1QL ist ein von Couchebase definierter Standard. Es wird versucht eine SQL-ähnliche Sprache bereit zu stellen welche mit Dokument-orientierten Daten nutzbar ist. Eine Beispiel-Query ist in Darstellung 1.3 dargestellt.

Darstellung 1.3 JSONiq Beispiel

```
SELECT name AS name
FROM people
WHERE age > 30
```

Kapitel 2

SQL++

SQL++ ist eine semi-strukturierte Query-Language die versucht relationale und JSON basierte Daten gleichzeitig zu unterstützen. Eines der Ziele von SQL++ ist die Vereinigung von verschiedenen Query-Languages. Dies wird durch eine starke Konfigurierbarkeit erreicht. Verschiedene Parameter können definiert werden die das Verhalten von Queries beeinflussen, was der Sprache erlaubt sich an die Semantik einer anderen Query-Language anzupassen. Version 4 von Couchbases N1QL ist als syntaktischer Zucker über SQL++ definierbar und AsterixDB unterstützt vollständiges SQL++. [5]

2.1 Datenmodell

Das Datenmodell von SQL++ soll sowohl SQL als auch JSON unterstützen. Darstellung 2.1 zeigt die Grammatik des Datenmodells. In der Grammatik ist zu erkennen das es sich um eine Erweiterung von JSON handelt.

Ein Wert ist ein Skalar, `null`, `missing` oder komplexer Wert. Ein Skalar ist ein einfacher Wert, wie eine Zahl oder eine Zeichenkette. Ein komplexer Wert ist eine Sammlung von Werten. SQL++ erweitert JSON um den `enriched_value`, dabei handelt es sich um einen skalaren Wert welcher mit einem Typnamen erweitert wurde. Dieser Typname beschreibt dass der Wert einen speziellen Typ besitzt und vereinbart wie dieser kodiert ist. Ein Beispiel für ein `enriched_value` ist ein Zeitstempel, dieser kann als eine speziell formatierte Zeichenkette dargestellt werden. Der Wert `timestamp('2016-01-01T12:00:00')` ist ein `enriched_value`.

Ein Tupel ist, wie in JSON, eine Menge von Paaren aus einer Zeichenkette und einem beliebigen Wert. Es wird in der Form `{` key_0 `:` $value_0$ `,…,` key_n `:` $value_n$ `}` dargestellt, wobei key_i eine Zeichenkette und $value_i$ der zugehörige Wert ist. Die Zeichenketten

Darstellung 2.1 SQL++ Daten-Grammatik

```
      <named_value> ::= <name> '::' <value>
            <value> ::= 'null'
                      | 'missing'
                      | <scalar_value>
                      | <complex_value>
    <complex_value> ::= <tuple_value>
                      | <collection_value>
     <scalar_value> ::= <primitive_value>
                      | <enriched_value>
  <primitive_value> ::= '"' <string> '"'
                      | <number>
                      | 'true'
                      | 'false'
   <enriched_value> ::= <type> '(' <primitive_value>
                           (',' <primitive_value>)* ')'
      <tuple_value> ::= '{' (<name> ':' <value>
                           (',' <name> ':' <value>)*)? '}'
 <collection_value> ::= <array_value>
                      | <bag_value>
      <array_value> ::= '[' (<value> (',' <value>)*)? ']'
        <bag_value> ::= '{{' (<value> (',' <value>)*)? '}}'
```

agieren als eindeutiger Bezeichner, womit keine Zeichenkette doppelt in einem Tupel vorkommen kann. Ein Tupel ist somit eine Abbildung die einem Namen je einem Wert zuordnet. Ein Erweiterung gegenüber JSON ist der `bag_value`. JSON sieht ausschließlich Arrays vor. Ein Array ist eine geordnete Sammlung von Werten welche eindeutig über ihren Index referenziert werden können, dargestellt als `[` $value_0$ `,…,` $value_n$ `]`. Ein Bag besitzt, im Gegensatz zum Array, keine Ordnung und wird dargestellt als `{{` $value_0$ `,…,` $value_n$ `}}`. SQL++ unterscheidet somit zwischen Sammlungen mit und ohne Ordnung.

Wie in JSON kann ein Wert `null` sein. SQL++ erlaubt außerdem das ein Wert `missing`, also fehlend, ist. Somit kann in der Query-Auswertung klar erkannt werden ob ein verlangter Wert Null ist oder nicht existiert. Diese Unterscheidung ist in SQL nicht nötig, da durch das Schema festgelegt ist welche Spalten in einer Tabelle existieren.

Die letzte Erweiterung gegenüber JSON ist der `named_value`. Dies ist ein Wert der durch einen im Datensatz einzigartigen Namen identifiziert wird. Die `named_value`s repräsentieren die Wurzelelemente einer Datenbank. Somit entsprechen sie den Schlüsseln der Dokumente in einem Document-Store oder den Tabellennamen einer SQL-Datenbank.

SQL

artist	
id	**name**
0	Mozart
1	Pink Floyd
2	Beethoven

SQL++

```
artist :: {{
  { id: 0, name: "Mozart" },
  { id: 1, name: "Pink Floyd" },
  { id: 2, name: "Beethoven" }
}}
```

Abbildung 2.1: SQL Tabelle als SQL++-Wert

2.1.1 Vergleich mit SQL

SQL++ erkennt dass relationale Daten als semi-strukturierte Daten darstellbar sind. Das SQL-Datenmodell ist ein Teil des SQL++-Datenmodells. Ein Tupel einer SQL-Tabelle kann als SQL++-Tupel dargestellt werden. Die Schlüssel im SQL++-Tupel entsprechen den Schlüsseln des SQL-Tupels welche die Namen der Spalten sind. Eine SQL-Tabelle entspricht einer Menge von Tupeln. In SQL++ entspricht eine SQL-Tabelle somit einem Array von Tupeln, wenn die Ordnung der Zeilen ignoriert wird entspricht eine Tabelle einem Bag von Tupeln. In Abbildung 2.1 ist eine SQL-Tabelle und der äquivalente SQL++-Wert dargestellt. Grundlegende Unterschiede zu SQL bestehen darin dass ein Bag oder Array heterogene Daten enthalten kann und ein Wert in SQL++ beliebig tief verschachtelt sein kann.

2.2 Queries

Eine Query in SQL ist immer eine SFW-Query (Select-From-Where-Query). In SQL++ kann eine Query eine SFW-Query oder eine Expression sein. Als Expression in SQL++ werden Ausdrücke bezeichnet welche einen Wert liefern, wie z.B. Literale, Funktionsaufrufe, Subqueries oder beliebige Kombinationen aus diesen. Der resultierende Wert einer Expression in SQL++ besitzt keine Einschränkungen, wobei SQL nur einen Skalar oder Null erlaubt. In der Grammatik (Darstellung 2.2) ist die hohe Kombinierbarkeit von Expressions und SFW-Queries zu erkennen.

2.2.1 Pfade

Eine Expression kann eine Pfad-Navigation sein. Dabei handelt es sich um eine Tupel-Navigation der Form `e.k` oder eine Array-Navigation der Form `e[i]`, wobei `e` eine eigene Expression ist. Wenn diese Expression zu einem Tupel ausgewertet wird, dann liefert

Darstellung 2.2 SQL++ Query-Grammatik

```
      <query> ::= <sfw_query>
                | <expr>
  <sfw_query> ::= <select_clause>
                  <from_clause>
                  ('WHERE' <expr>)?
                  ('GROUP BY' <expr> ('AS' <var>)?
                    (',' <expr> ('AS' <var>)*)+ )?
                  ('HAVING' <expr>)?
                  (('UNION' | 'INTERSECT' | 'EXCEPT') 'ALL'? <sfw_query>)?
                  ('ORDER BY' <expr> ('ASC' | 'DESC')?
                    (',' <expr> ('ASC' | 'DESC')?)* )?
                  ('LIMIT' <expr>)?
                  ('OFFSET' <expr>)?
       <expr> ::= '(' <sfw_query> ')'
                | <named_value>
                | <var>
                | <expr> '.' <attr_name>
                | <expr> '[' <expr> ']'
                | <function_name> '(' (<expr> (',' <expr>)*)? ')'
                | '{' (<attr_name> ':' <expr>
                    (',' <attr_name> ':' <expr>)*)? '}'
                | '[' (<expr> (',' <expr>)*)? ']'
                | '{{' (<expr> (',' <expr>)*)? '}}'
                | <value>
```

`e.k` den Wert des Paares mit dem Schlüssel `k` aus diesem Tupel. Wenn sie zu einem Array ausgewertet wird, dann liefert `e[i]` das i-te Element aus diesem Array.

Durch die Kombinierbarkeit von Expressions können Pfad-Navigationen beliebig verkettet werden. Da kein Schema existiert gibt es keine Garantie dass die entsprechende Pfad-Navigation anwendbar ist. Außerdem muss festgelegt werden was bei der Nicht-Existenz eines gesuchten Unterelementes passiert. SQL++ sieht Konfigurationsparameter vor welche das Verhalten einer Navigation auf fehlerhafte Typen (`type_mismatch`) oder referenzieren von nicht existierenden Elementen (`absent`) beschreibt. Jeder dieser Parameter kann auf `error`, `null` oder `missing` gesetzt werden, was einen Fehler auslöst oder `null`, beziehungsweise `missing`, zurück gibt.

2.2.2 Variablenbindungen

Jeder Query und Subquery wird in einer Umgebung Γ ausgewertet. Die Umgebung besteht aus einer Konfigurations- und Variablenumgebung (Γ_c und Γ_b). In der Konfigurationsumgebung sind alle Einstellungen für den aktuellen Query enthalten (später genauer

erklärt). Die Variablenumgebung ist strukturell ein SQL++-Tupel, in diesem werden Werte an einzigartige Variablennamen (Schlüssel des Tupels) gebunden. $\Gamma_b \vdash Q \rightarrow V$ bedeutet dass der Query Q ausgewertet in Variablenumgebung Γ_b den Wert V erzeugt, alle Variablen in Q werden dabei mit dem entsprechenden Wert aus Γ_b belegt. Im weiteren ist mit Umgebung, wenn nicht genauer spezifiziert, die Variablenumgebung gemeint. Die Binding-Tupel b und b' können verkettet werden. Das dabei enstehenden Tupel $b||b'$ enthält alle Namen aus b und b'. Wenn ein Name in b und b' vorkommt wird nur der aus b übernommen, womit die Einzigartigkeit der Namen weiterhin gegeben ist.

Ein Query wird zu Beginn in einer Umgebung ausgewertet die den benannten Werten der Datenbank (also den Tabellen oder Dokumenten) entspricht. Die `FROM` Klausel eines SFW-Queries erzeugt neue Variablenbindungen die von den weiteren Klauseln des SFW=Queries verändert werden. Subqueries des SFW-Queries werden in einer erweiterten Umgebung ausgeführt.

2.2.3 SFW-Queries

SFW-Queries in SQL++ bestehen aus einer Menge von Klauseln welche in einer festen Reihenfolge ausgewertet werden.

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. ORDER BY
6. LIMIT / OFFSET
7. SELECT

Die Auswertung der Klauseln gleicht einer Pipeline. Jede Klausel produziert eine Menge (Bag oder Array) von Binding-Tupeln. Die Ausgabe einer Klausel sei $B^{out}_{Klausel}$ und die Eingabe $B^{in}_{Klausel}$. Die Ausgabe einer Klausel ist äquivalent zur Eingabe der nächsten existierenden Klausel (z.B. $B^{out}_{FROM} = B^{in}_{WHERE}$). `FROM` besitzt keine Eingabe und ist nur von der Umgebung des SFW-Queries (Γ_b) abhängig. Die Ausgabe von `SELECT` ist der finale Wert des SFW-Queries. Das allgemeine Verhalten ist somit das jede Klausel einen Bag von Tupeln als Eingabe erhält, die entsprechende Operation ausführt (inklusive der Auswertung der gegebenen Expressions) und einen Bag von Tupeln ausgibt. `ORDER BY` erzeugt ein Array als Ausgabe (was bedeutet das die Reihenfolge der Tupel relevant ist). Die nachfolgenden Klauseln welche ein Array als Eingabe erhalten produzieren selbst Arrays, damit die von `ORDER BY` eingeführte Ordnung nicht verloren geht.

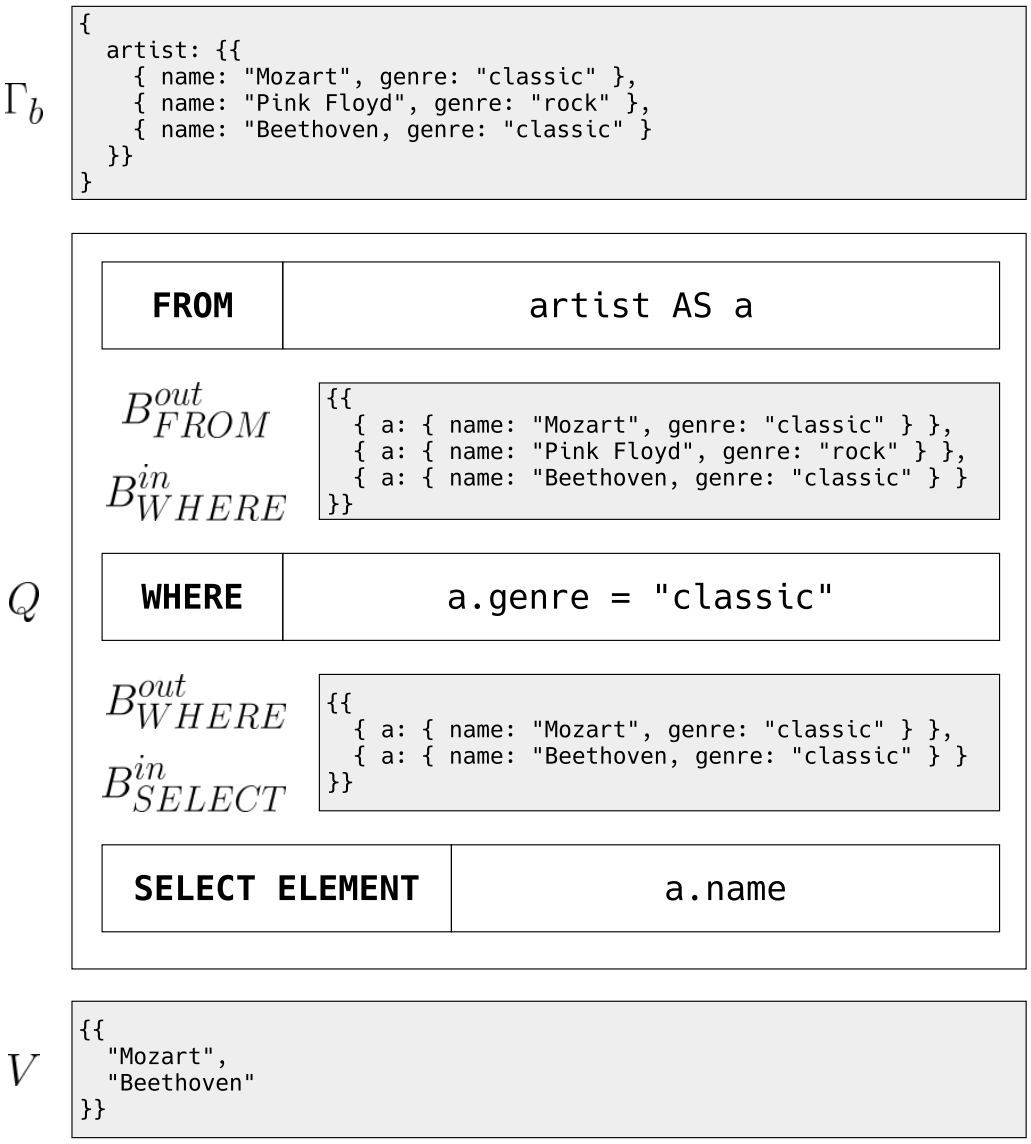


Abbildung 2.2: SFW-Query Auswertung

In Abbildung 2.2 ist die Ausführung eines Beispiel-SFW-Queries ($\Gamma_b \vdash Q \to V$) dargestellt. Die Query wird in der Umgebung Γ_b ausgewertet, wenn die Query kein Subquery ist entspricht diese Umgebung den Daten in der Datenbank. `FROM` besitzt die Expression `artist`, diese Expression wird in Γ_b ausgewertet. $\Gamma \vdash$ `artist` liefert den entsprechenden Bag aus der Umgebung. `FROM` generiert für jeden Wert in diesem Bag ein Binding-Tupel welches den Schlüssel `a` (wie in `... AS a` angegeben) mit dem aktuellen Wert belegt. Somit besteht B^{out}_{FROM} und B^{in}_{WHERE} aus 3 Binding-Tupeln.

`WHERE` wertet für jedes Tupel die Expression `a.genre = "classic"` aus. Die Expression wird für das Binding-Tupel b in der Umgebung $b||\Gamma_b$ ausgewertet. Die erweiterte Umgebung für das erste Binding-Tupel ist somit `{ a: { name: "Mozart", genre: "classic" }, artist: {{ ... }} }`, womit die Expression den Wert `true` liefert. Die Ausgabe B^{out}_{WHERE} ergibt sich aus allen Tupeln für welche die Expression `true` liefert (2 Tupel).

`SELECT ELEMENT` generiert einen Bag (oder Array) in dem die Werte der Expression (hier `a.name`) für jedes Tupel in B^{in}_{SELECT} enthalten sind. Dies ist der finale Wert V des Queries Q.

2.2.4 SELECT-Klausel

Darstellung 2.3 SELECT-Klausel

```
<select_clause> ::= 'SELECT ELEMENT' <expr>                          (1)
                  | 'SELECT ATTRIBUTE' <expr> ':' <expr>             (2)
                  | 'SELECT' <expr> ('AS' <attr_name>)?
                      (',' <expr> ('AS' <attr_name>)?)*              (3)
```

Die Ausgabe einer `SELECT`-Klausel in SQL++ ist, im Gegensatz zu SQLs `SELECT`, nicht nur auf Mengen von Tupeln beschränkt. Es können Mengen von beliebigen Werten ausgegeben werden, dies beinhaltet tief verschachtelte Werte sowie einzelne Skalare. Die Ausgabe-Menge muss außerdem nicht homogen sein. Zusätzlich ist es möglich ein einzelnes Tupel auszugeben dessen Attribute von den Eingabedaten abhängen.

- `SELECT ELEMENT` e (Darstellung 2.3 (1)) gibt ein Sammlung (Bag oder Array) von Werten aus. Für jedes Binding-Tupel b in B^{in}_{SELECT} wird ein Wert v mit $b||\Gamma \vdash e \to v$ generiert. Diese Werte bilden die Ausgabe-Sammlung. Diese Sammlung ist ein Array wenn die Eingabe (B^{in}_{SELECT}) ebenfalls ein Array ist.

  Die Klausel

  ```
  SELECT ELEMENT x+1 FROM [1,2,3] AS x
  ```

  produziert somit die Ausgabe

```
[2,3,4]
```

- `SELECT ATTRIBUTE` e_n `:` e_v (Darstellung 2.3 (2)) gibt ein Tupel aus dessen Key-Value-Paare von der Eingabe abhängen. Für jedes Binding-Tupel b in B^{in}_{SELECT} wird ein Key-Value-Paar $n : v$ mit $b||\Gamma \vdash e_n \to n$ und $b||\Gamma \vdash e_v \to v$ generiert. Diese Paare bilden das Ausgabe-Tupel. n muss dabei ein String sein, sonst tritt ein Fehler auf.

  Die Klausel

  ```
  SELECT ATTRIBUTE x.name : x.born
  FROM [ { name: "Mozart",    born: 1756 },
         { name: "Beethoven", born: 1770 } ] AS x
  ```

  produziert somit die Ausgabe

  ```
  { Mozart: 1756, Beethoven: 1770 }
  ```

- `SELECT` e_0 `AS` a_0 `,` … `,` e_n `AS` a_n (Darstellung 2.3 (3)) ist syntaktischer Zucker für einen Spezialfall von `SELECT ELEMENT` um SQL-artige Syntax zu erlauben. Die Klausel ist identisch zu `SELECT ELEMENT {` a_0 `:` e_0 `,` … `,` a_n `:` e_n `}`. Wenn kein Name (`AS` a) gegeben ist wird versucht einen Namen aus e abzuleiten. Wenn e ein Pfad ist, ist `SELECT` $e'.n$ äquivalent zu `SELECT` $e'.n$ `AS` n.

2.2.5 FROM-Klausel

Darstellung 2.4 FROM-Klausel

```
<from_clause> ::= 'FROM' <from_item>
  <from_item> ::= <expr> 'AS' <var> ('AT' <var>)?                          (1)
                | <expr> 'AS' '{' <var> ':' <var> '}'                      (2)
                | <from_item> ('INNER'|'LEFT' 'OUTER'?) 'CORRELATE'?
                    <from_item>                                            (3)
                | <from_item> 'FULL' 'OUTER'? CORRELATE? <from_item>
                    ON <expr>                                              (4)
                | <from_item> ',' <from_item>                              (5)
                | <from_item> ('INNER'|'LEFT'|'RIGHT'|'FULL') 'JOIN'
                    <from_item> 'ON' <expr>                                (6)
                | ('INNER'|'OUTER') 'FLATTEN'
                    '(' <expr> AS <var> ',' <expr> 'AS' <var> ')'          (7)
```

Die FROM-Klausel ist der logische Ausgangspunkt eines SFW-Queries und produziert die initialen Binding-Tupel. `FROM` leitet die Klausel ein und ist gefolgt von einem `from_item`, welches rekursiv definiert ist.

- e `AS` n `AT` i (Darstellung 2.4 (1)) erwartet dass die Expression e ein Bag oder Array ist. Für jedes Element in e wird ein Binding-Tupel generiert. In diesem Binding-Tupel ist der Name n auf den entsprechenden Wert aus e abgebildet. Wenn `AT` i vorhanden ist wird der Name i auf den Index des aktuellen Wertes im Array e abgebildet.

  Die Klausel

  ```
  FROM ["a", "b"] AS x AT y
  ```

  produziert somit die Binding-Tupel

  ```
  {{ { x:"a", y:0 }, { x:"b", y:1 } }}
  ```

- e `AS {` a `:` b `}` (Darstellung 2.4 (2)) erwartet dass die Expression e ein Tupel ist. Für jedes Key-Value-Paar k `:` v in e wird ein Binding-Tupel der Form `{` a`:`k`,` b`:`v `}` generiert.

  Die Klausel

  ```
  FROM { a:1, b:2 } AS { x:y }
  ```

  produziert somit die Binding-Tupel

  ```
  {{ { x:"a", y:1 }, { x:"b", y:2 } }}
  ```

- l `INNER CORRELATE` r (Darstellung 2.4 (3)) kombiniert die `from_item`s l und r. Dabei kann r Namen benutzen welche in l definiert wurden. Sei Γ die gegebene Umgebung und B^l die von l generierten Binding-Tupel, welche der Ausgabe von `FROM` l in der Umgebung Γ entsprechen. Für jedes Binding-Tupel $b_l \in B^l$ werden die Binding-Tupel B^r von r generiert, welche der Ausgabe von `FROM` r in der Umgebung $b_l||\Gamma$ entsprechen. Jedes Binding-Tupel $b_r \in B^r$ bildet verkettet mit b $(b||b_r)$ ein Binding-Tupel der finalen Ausgabe der Klausel.

  Die Klausel

  ```
  FROM [ { n:"a", nr:[1,2] },
         { n:"b", nr:[3] },
         { n:"c": nr:[] } ] AS x INNER CORRELATE x.nr AS y
  ```

  produziert somit die Binding-Tupel

  ```
  {{ { x: { n:"a", nr:[1,2] }, y:1 },
     { x: { n:"a", nr:[1,2] }, y:2 },
     { x: { n:"b", nr:[3]   }, y:3 } }}
  ```

- l `LEFT OUTER CORRELATE` r (Darstellung 2.4 (3)) verhält sich ähnlich zu `INNER CORRELATE`. Jedoch wird wenn r eine leere Menge liefert dennoch ein Binding-Tupel generiert. Der fehlende Wert wird dabei, je nach Konfiguration, auf `null` oder `missing` gesetzt.

  Die Klausel

  ```
  FROM [ { n:"a", nr:[1,2] },
         { n:"b", nr:[3] },
         { n:"c": nr:[] } ] AS x LEFT OUTER CORRELATE x.nr AS y
  ```

  produziert somit die Binding-Tupel

  ```
  {{ { x: { n:"a", nr:[1,2] }, y: 1 },
     { x: { n:"a", nr:[1,2] }, y: 2 },
     { x: { n:"b", nr:[3]   }, y: 3 },
     { x: { n:"c", nr:[]    }, y: missing } }}
  ```

- l `FULL OUTER CORRELATE` r `ON` c (Darstellung 2.4 (4)) verhält sich identisch zu SQLs Full-Join. Die `from_item`s l und r werden unabhängig voneinander ausgewertet und liefern die Binding-Tupel B^l und B^r. Für alle $b_l \in B^l$ und $b_r \in B^r$, mit welchen $b_l||b_r||\Gamma \vdash c$ zu `true` ausgewertet wird, wird ein Binding-Tupel $b_l||b_r$ der finalen Ausgabe der Klausel generiert. Wenn ein b_l oder b_r nicht verwendet wurde, wird ein Binding-Tupel mit diesem generiert bei welchem der Wert des Partners auf `null` oder `missing` gesetzt ist.

  Die Klausel

  ```
  FROM [1,2] AS x FULL OUTER CORRELATE [1,2] AS y ON x > y
  ```

  produziert somit die Binding-Tupel

  ```
  {{ { x:1, y:missing }, { x:2, y:1 }, { x:missing, y:2 } }}
  ```

- l, r (Darstellung 2.4 (5)) ist syntaktischer Zucker für l `INNER CORRELATE` r bei dem r keine Variablen aus l benutzt.

  Die Klausel

  ```
  FROM ["a","b"] AS x, [1,2] AS y
  ```

  produziert somit die Binding-Tupel

  ```
  {{ { x: "a", y:1 },
     { x: "a", y:2 },
     { x: "b", y:1 },
  ```

```
{ x: "b", y:2 } }}
```

- l `INNER|LEFT|RIGHT|FULL JOIN` r `ON` c (Darstellung 2.4 (6)) ist syntaktischer Zucker für verschiedene `CORRELATE` Klauseln und existiert für Kompatibilität und Ähnlichkeit zu SQL.

```
* FROM l AS x INNER JOIN r AS y ON c
⇒ FROM l AS x INNER CORRELATE
      (SELECT ELEMENT y FROM r AS y WHERE c) AS y

* FROM l AS x LEFT JOIN r AS y ON c
⇒ FROM l AS x LEFT CORRELATE
      (SELECT ELEMENT y FROM r AS y WHERE c) AS y

* FROM l AS x RIGHT JOIN r AS y ON c
⇒ FROM r AS y LEFT JOIN  l AS x ON c

* FROM l AS x FULL JOIN r AS y ON c
⇒ FROM l AS x FULL CORRELATE r AS y ON c
```

- `INNER|OUTER FLATTEN (` l `AS` x `,` r `AS` y `)` (Darstellung 2.4 (7)) ist syntaktischer Zucker für l `AS` x `INNER|OUTER CORRELATE` r `AS` y. Dabei ist r eine Pfad-Navigation in x. Flatten, auch als Unnest bezeichnet, ist eine in einigen NoSQL-Datenbanken vorhandene Klausel welche verschachtelte Arrays entpackt.

  Die Klausel

```
FROM INNER FLATTEN ([ { n:"a", nr:[1,2] },
                      { n:"b", nr:[3] } ] AS x, x.nr AS y)
```

  produziert somit die Binding-Tupel

```
{{ { x: { n:"a", nr:[1,2] }, y: 1 },
   { x: { n:"a", nr:[1,2] }, y: 2 },
   { x: { n:"b", nr:[3]   }, y: 3 } }}
```

2.2.6 Andere Klauseln

- `WHERE` c entfernt alle Binding-Tupel für die die Expression c nicht `true` ergibt.
- `ORDER BY` e_0 `ASC|DESC , … ,` e_n `ASC|DESC` sortiert die Binding-Tupel nach dem Wert der Expression e_i, wenn dieser identisch ist wird der Wert von e_{i+1} genutzt.

- `OFFSET` e ignoriert die ersten n Binding-Tupel. n ergibt sich aus e und muss eine natürliche Zahl sein.

- `LIMIT` e reduziert die Anzahl der Binding-Tupel auf die Anzahl welche sich aus e ergibt. e muss eine natürliche Zahl ergeben.

- `GROUP BY` e_0 `AS` v_0 `,` … `,` e_n `AS` v_n gruppiert Binding-Tupel. Es werden Gruppen gebildet so dass jede Expression e_0, … , e_n für alle Tupel einer Gruppe identische Werte produziert. Jede Gruppe generiert ein Binding-Tupel der Ausgabe welche den Schlüssel `group` auf einen Bag mit allen zu dieser Gruppe gehörigen Binding-Tupel abbildet. Wenn `AS` v_i gegeben ist wird in jedem Binding-Tupel der Ausgabe der Schlüssel v_i auf den Wert von e_i abgebildet.

  Die Klausel `GROUP BY genre AS g` für die Binding-Tupel $B^{in}_{GROUPBY}$

  ```
  {{ { name: "Mozart", genre: "classic" },
     { name: "Pink Floyd", genre: "rock" },
     { name: "Beethoven", genre: "classic" } }}
  ```

  produziert somit die Binding-Tupel $B^{out}_{GROUPBY}$

  ```
  {{ { g: "classic", group: {{
       { name: "Mozart",    genre: "classic" },
       { name: "Beethoven", genre: "classic" } }} },
     { g: "rock", group: {{
       { name: "Pink Floyd", genre: "rock" } }} } }}
  ```

  Aggregatfunktionen in `SELECT`, `HAVING`, und `ORDER BY` besitzen syntaktischen Zucker um SQLs verhalten nachzustellen. Die Expression `f(e)`, wenn `f` eine Aggregatfunktion (`sum`, `avg`, …) und `e` eine Expression mit Namen aus der FROM-Klausel ist, wird übertragen zu `f(SELECT e' FROM group AS p)`, wobei `e'` die Expression `e` ist mit allen Vorkommen der Variable v_i ersetzt durch $p.v_i$.

- `HAVING` c verhält sich identisch zu `WHERE` c, erlaubt jedoch Aggregatfunktionen in c.

- l `UNION|INTERSECT|EXCEPT` (`ALL`) r verbindet die Ausgaben von den kompletten SFW-Queries l und r. Identisch zum Verhalten in SQL werden beide SFW-Queries ausgeführt und die Ergebnisse mit der entsprechenden Mengenoperation verbunden. Duplikate werden entfernt außer wenn `ALL` gegeben ist.

  - `UNION` bildet die Vereinigung, also alle Elemente aus beiden Mengen.
  - `INTERSECT` bildet den Durchschnitt, also alle Elemente die in beiden Mengen vorkommen.

– `EXCEPT` bildet die Differenz, also alle Elemente aus der linken Menge die nicht in der rechten vorkommen.

2.2.7 Konfiguration

SQL++ besitzt eine Vielzahl von Konfigurationsparametern welche die Query-Ausführung beeinflussen. Da SQL++ versucht mit anderen Query-Languages kompatibel zu sein ist diese hohe Konfigurierbarkeit nötig. Queries und Subqueries können außerdem mit Konfig-Annotationen versehen werden um die Semantik in sehr spezieller Weise zu manipulieren. In der im Laufe dieser Arbeit implementierten Query-Language sind diese Annotationen nicht vorgesehen. Viele der Konfigurationsparameter werden im Quellcode bereit gestellt, womit diese später in der Entwicklung den Ansprüchen angepasst werden können. Das Ziel ist nicht den kompletten Umfang von SQL++ zu implementieren, sondern eine angemessene Variante umzusetzen.

Konfigurationsparameter sind unter anderen:

- Resultat von fehlgeschlagenen Pfad-Navigationen (`null`, `missing`, Fehler)
- Ergebnis von `=` zwischen allen Kombinationen von `null`, `missing` und Boolean

2.2.8 Algebra und Funktionen

Es ist eine Algebra zwischen Expressions vorgesehen, diese ist in SQL++ jedoch nicht genau spezifiziert und beruft sich stattdessen auf die Algebra von SQL. Die vorhandenen Funktionen sind ebenfalls nicht genau festgelegt. In dieser Arbeit werden mathematische Operatoren (`+`,`-`,`*`,`/`,`%`) zwischen Zahlen, boolesche Operatoren (`AND`, `OR`, …), Vergleichsoperatoren (`<`,`>`,`=`,`<>`) und Funktionsaufrufe implementiert.

Kapitel 3

Language-Processor

Es ist ein Programm zu verfassen welches Queries im zuvor definierten Format entgegen nimmt, diese ausführt und die resultierenden Daten zurück gibt. Eine Query ist als Quellcode eines Programms zu verstehen. Somit handelt es sich bei dem gewünschten Programm um einen Language-Processor. Bei einem Language-Processor kann es sich um einen Compiler, Interpreter oder Hybrid (Abbildung 3.1) handeln. [1]

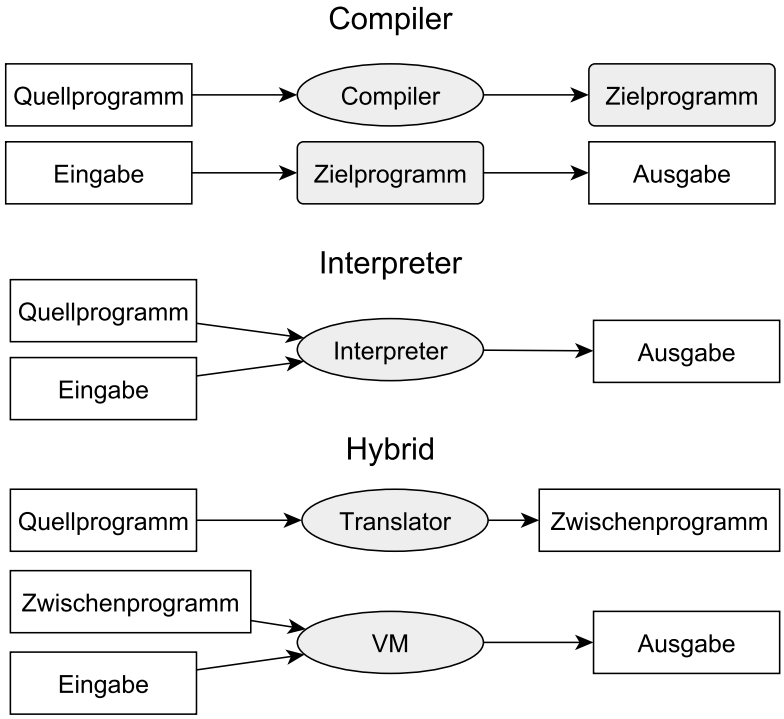


Abbildung 3.1: Language-Processor Typen

Reine Compiler, welche alleinstehenden Maschinencode generieren, sind untypisch in einem DBMS. Da die compilierte Query im Kontext der Datenbank ausgeführt wer-

den muss, ist das Erzeugen eines allein ausführbaren Zielprogramms nicht geeignet. Interpreter und Hybride sind weit verbreitet.

PostgreSQL nutzt einen Interpreter [6]. Dieser übersetzt eine Query in einen internen Query-Tree. Dieser wird anschließend optimiert und in einen Query-Plan umgewandelt, welcher vom Executor ausgeführt wird.

SQLite nutzt einen Hybrid [8]. Dabei wird eine Query in ein Programm in einer Maschinencode ähnlichen Sprache übersetzt. Dieses Programm wird anschließend von einer virtuellen Maschine, im Kontext einer Datenbank-Instanz, ausgeführt. Es wird argumentiert dass diese Variante eine klarere Trennung zwischen Frontend (Parsen und Generieren) sowie Backend (Ausführung und Berechnung des Ergebnisses) der Query-Verarbeitung bietet.

Beide Varianten sind gut für die Implementierung einer Query-Language geeignet. Wie an den Beispielen *SQLite* und *PostgreSQL* zu sehen, finden beide Varianten in etablierten DBMS Einsatz. Die Übersetzung einer Query in ein Programm in einem selbst definierten Bytecode, wie es bei einem Hybrid stattfindet, bietet gewisse Vorteile. So kann isoliert an der Übersetzung in den Bytecode gearbeitet werden, was starke Optimierungen in der Codegenerierung erlaubt. Gleichzeitig kann auch der Executor isoliert entwickelt werden. Bei gleichbleibender Spezifikation des Bytecodes ist selbst ein Austauschen der Module einfach. Jedoch stellt diese Variante auch einen größeren Aufwand dar und bietet ohne separate Entwicklerteams kaum Vorteile. Somit fällt die Wahl in dieser Arbeit auf einen reinen Interpreter.

3.1 Interpreter-Strategie

Wie in Abbildung 3.1 dargestellt erhält ein Interpreter ein Quellprogramm sowie eine Eingabe, verarbeitet diese und generiert eine Ausgabe. Im Kontext eines DBMS ist das Quellprogramm eine Query und die Ausgabe das Resultat dieser Query. Die Eingabe kann als Instanz einer Datenbank verstanden werden auf welche sich die Query bezieht.

Interpreter folgen generell eine dieser Strategien:

- Parsen und direktes Ausführen des Verhaltens
- Parsen in eine internen Repräsentation und Ausführen dieser

Parsen bezeichnet das syntaktische Verarbeiten und zerlegen der Zeichenkette, welche das Quellprogramm darstellt, unter Nutzung der Grammatik. Ein direktes Ausführen beim Parsing ist bei komplexen Grammatiken kaum geeignet. Ein Trennen von Parsing

und Ausführen ist vorteilhaft und wird, außer bei trivialen Grammatiken, generell vorgenommen.

Der erste Schritt des Interpreters ist die Query in einer interne Repräsentation umzuwandeln. Die interne Repräsentation spiegelt die Grammatik der Sprache wieder und besitzt somit eine Baumstruktur. Sie wird als *Abstract-Syntax-Tree* (AST) bezeichnet.

Anschließend folgt die Ausführung der Query anhand des AST. Dabei kann der AST direkt zur Ausführung genutzt werden oder in einen andere Darstellung umgewandelt werden. Die Grammatik von SQL++ ist stark an der Ausführung orientiert. Die Pipeline-artige Funktionsweise von Select-From-Where Expressions sowie die Evaluierung von verschachtelten Expressions in der Umgebung der enthaltenden Expression legen es nahe den AST direkt zur Ausführung zu nutzen.

Somit muss ein AST, welcher die Struktur einer SQL++-Query widerspiegelt, definiert werden. Anschließend muss ein Parser, welcher den AST aus einem Query generiert, sowie ein Executor, welcher den AST ausführt und ein Ergebnis liefert, programmiert werden.

Kapitel 4

Abstract-Syntax-Tree

Das Format des AST sollte sich an der Grammatik von SQL++ orientieren (Darstellung 2.2, 2.3, 2.4). Außerdem muss die logische Ausführung im AST widergespiegelt werden.

4.1 Vererbung gegen Vereinigung

Das Zentrale Element ist die Expression, welche ein abstrakter Typ ist. Eine Expression muss somit in der Lage sein verschiedene Typen darzustellen.

In Objektorientierter-Programmierung, und C++ speziell, existieren zwei Möglichkeiten eine solche Beziehung darzustellen: Vererbung oder Vereinigung (Union). Mit Vererbung ist eine Expression ein abstrakter Basistyp, von welcher andere Typen ableiten. Bei der Nutzung von Vereinigung ist eine Expression ein Union-Typ aller implementierenden Typen, im Speicher einer Expression wird somit einer der aufgelisteten Typen gehalten. Neben der Benutzung unterscheiden sich beide Varianten in der Darstellung im Speicher. Da die implementierenden Typen unterschiedlich viel Speicher benötigen muss eine Menge (z.B. ein Array) von Expressions die mit Vererbung implementiert sind stets durch Pointer zu extern allozierten Speicher umgesetzt werden. Eine Menge Expressions welche mit Union implementiert sind kann hingegen alle Elemente lokal speichern, verschwendet jedoch Speicher da ein Union-Typ immer so groß sein muss wie der größte enthaltende Typ. In Abbildung 4.1 ist dieser Vergleich anhand eines Arrays von Expressions visualisiert. Da kontinuierlicher Speicher effizienter zu lesen ist ist die Union-Variante effizienter, obwohl sie ungenutzten Speicher enthält. Die Entscheidung lässt sich nur schwer objektiv treffen. Da die genutzte Parsing-Bibliothek ([2]) `variant` bevorzugt (ein typsicherer Union) wird die Union-Variante genutzt.

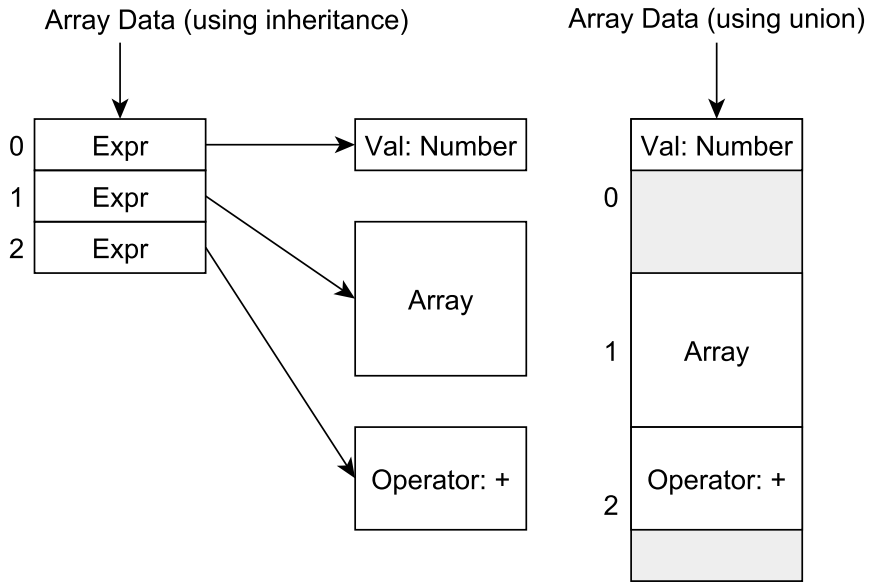


Abbildung 4.1: Vererbung gegen Union

4.2 Expression

Darstellung 4.1 zeigt die Deklaration einer Expression im AST. Eine Expression ist als `variant` über alle möglichen Untertypen umgesetzt. Typen wie `InfixOp` oder `SfwQuery` enthalten weitere Expressions und sind somit rekursiv definiert. Da ein Typ kein Teil von sich selbst sein kann ist es nicht möglich die rekursiven Typen direkt in einer Expression zu halten. Aus diesem Grund wird die Hilfsklasse `forward_ast` genutzt. Diese ist als Pointer implementiert der auf das entsprechende Element, welches extern alloziert wurde, verweist. Tuple, Bag und Array sind als dynamische Container implementiert, womit sie ihre Elemente nicht lokal halten und nicht in einen `forward_ast` gepackt werden müssen.

Darstellung 4.1 AST: Expression

```
struct Expr : variant<Var,
                      Tuple,
                      Bag,
                      Array,
                      Function,
                      model::Value,
                      forward_ast<SfwQuery>,
                      forward_ast<TupleNav>,
                      forward_ast<ArrayNav>,
                      forward_ast<InfixOp>,
                      forward_ast<PrefixOp>> { };
```

4.3 Variablen

Wie in Darstellung 4.2 zu sehen ist sind Variablen als einfache Zeichenketten implementiert. Diese Zeichenkette ist der Name der Variable. Dem Variablennamen eine Bedeutung zuzuweisen erfolgt während der Ausführung der Query.

Darstellung 4.2 AST: Variable

```
struct Var : string { };
```

Die Variable im AST repräsentiert eine Variable oder ein `named_value` in der Grammatik. Ob ein Name in einer Query eine Variable oder ein Element in der Datenbank bezeichnet ist nur abhängig davon ob eine Variable mit entsprechendem Namen zuvor eingeführt wurde. In der Query

```
SELECT a, b FROM table AS a
```

ist `a` eine Variable und `b` ein `named_value`. Wenn diese Query jedoch als Sub-Query der Form

```
SELECT ELEMENT
  (SELECT a, b FROM table1 AS a)
FROM table2 AS b
```

ausgewertet wird, ist auch `b` eine Variable. Der `named_value` wurde von einem Variablennamen überdeckt. Somit ist es naheliegend die Unterscheidung zwischen Variable und `named_value` in der Ausführung statt im AST zu treffen.

4.4 Komplexe Werte

Tuple, Bag und Array werden, wie in Darstellung 4.3 dargestellt, durch Container der Standardbibliothek implementiert. Es ist zu bemerken dass es sich bei diesen nicht um Werte handelt, sondern Konstruktoren welche einen Tupel, Bag oder Array generieren. `Bag` und `Array` unterscheiden sich nur in der Ausführung und sind als `vector` (ein dynamisches Array) von Expressions implementiert. `Tuple` ist durch eine `map` (eine Abbildung) implementiert und nutzt eine Zeichenkette als Key sowie eine Expression als Wert. Es ist zu erkennen dass die Namen in Tupeln keine Expression sein können, sie müssen also direkt im Query gegeben sein und nicht aus Unterqueries zur Laufzeit generiert werden.

Darstellung 4.3 AST: komplexe Werte

```
struct Tuple : map<string, Expr> { };
struct Bag : vector<Expr> { };
struct Array : vector<Expr> { };
```

4.5 Pfad-Navigation

In Darstellung 4.4 sind Knoten dargestellt welche die Pfad-Navigationen im AST bilden. Beide bestehen aus einer Basis-Expression sowie einem Argument. Das Argument ist bei einer Tupel-Navigation eine Zeichenkette welche einem Schlüssel im Tupel entspricht. In der Array-Navigation ist das Argument der Index welcher einer Position im Array entspricht. Der Index ist als Expression implementiert, da komplexe Operationen in Array-Navigationen erlaubt sind, in der Ausführung muss diese Expression den korrekten Typ liefern.

Darstellung 4.4 AST: Pfad-Navigation

```
struct TupleNav { Expr base; string key; };
struct ArrayNav { Expr base; Expr idx; };
```

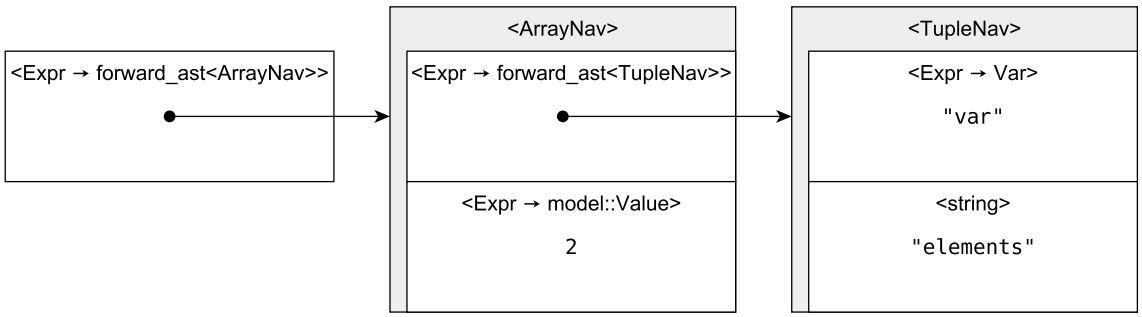


Abbildung 4.2: Speicherdarstellung des AST einer Pfad-Navigation

Pfad-Navigationen beinhalten eine vollständige Expression, weshalb sie nicht direkt als Unterelement einer Expression definiert werden können. Sie sind durch die `forward_ast`-Hilfsklasse als Pointer implementiert. Die Speicherstruktur des AST der Expression `var.elements[2]` ist in Abbildung 4.2 dargestellt.

4.6 Funktionen

Eine Funktion (Darstellung 4.5) besteht aus einem Funktionsnamen sowie einer beliebigen Anzahl von Argumenten, welche als `vector<Expr>` (dynamisches Array von Expressions) implementiert sind. Bei der Ausführung muss die entsprechende Funktion prüfen ob die korrekte Anzahl von Argumenten gegeben ist.

Darstellung 4.5 AST: Funktion

```
struct Function {
  string name;
  vector<Expr> arguments;
};
```

4.7 Operatoren

Die Algebra wird mit `PrefixOp` und `InfixOp` (Darstellung 4.6) umgesetzt. Die Details der Algebra sind in der SQL++-Spezifikation nicht gegeben. `PrefixOp` besteht aus einem Prefix-Operator, wie Negation (`-`), und einer Expression auf welche der Operator angewandt wird. `InfixOp` besteht aus einem Infix-Operator sowie einer linken und rechten Expression. Infix-Operatoren sind Rechen-Operatoren (`+ - * / %`), Vergleichsoperatoren (`< <= > >= = !=`) und logische Operatoren (`AND OR NOT`). Berechnungen werden somit als Baum dargestellt. Der Operatoren-Vorrang muss beim Parsing beachtet werden.

Darstellung 4.6 AST: Operatoren

```
struct PrefixOp { PrefixOperator op; Expr val; };
struct InfixOp { Expr left; InfixOperator op; Expr right; };
enum PrefixOperator { neg };
enum InfixOperator  { plus, minus, mul, div, modulo,
                      lt, le, gt, ge, eq, neq,
                      bool_and, bool_or, bool_not };
```

4.8 SFW-Query

Eine SFW-Query (Darstellung 4.7) besteht aus allen existierenden Klauseln. Nur `SELECT` und `FROM` sind in jeder SFW-Query enthalten. Alle weiteren Klauseln sind Optional, was mittels der `optional` Hilfsklasse implementiert ist.

Darstellung 4.7 AST: SFW-Query

```
struct SfwQuery {
  Select select;
  From from;
  optional<Where> where;
  optional<GroupBy> group_by;
  optional<Having> having;
  optional<OrderBy> order_by;
  optional<Limit> limit;
  optional<Offset> offset;
};
```

4.8.1 SELECT

Die `SELECT`-Klausel (Darstellung 4.8) besitzt zwei mögliche Varianten. Es kann sich um ein `SELECT ELEMENT` oder `SELECT ATTRIBUTE` handeln. `SELECT ELEMENT` besitzt eine einzelne Expression als Argument. `SELECT ATTRIBUTE` besitzt ein Paar aus Attributnamen und -wert. Der Attributname ist an dieser Stelle ebenfalls eine Expression, da er während der Ausführung generiert wird.

Darstellung 4.8 AST: SELECT-Klausel

```
struct Select : variant<SelectElement,
                        SelectAttribute> { };
struct SelectElement {
  Expr expr;
};
struct SelectAttribute {
  Expr attr;
  Expr value;
};
```

In der Grammatik existiert außerdem die an SQL orientierte Expression-Liste (`SELECT expr1 AS name1, expr2 AS name2, ...`). Diese ist jedoch als Syntaktischer-Zucker für `SELECT ELEMENT` definiert (`SELECT ELEMENT { name1:expr1, name2:expr2, ... }`) und benötigt somit keinen eigenen Typ im AST.

4.8.2 FROM

Die `FROM`-Klausel (Darstellung 4.9) besitzt durch die verschiedenen Join-Varianten eine hohe Kombinierbarkeit und ist aus diesem Grund rekursiv definiert. Die `FROM`-Klausel kann somit einen Baum bilden.

Darstellung 4.9 AST: FROM-Klausel

```
struct From : variant<FromEmpty, FromCollection, FromTuple,
                      FromInner, FromLeft, FromFull> { };

struct FromEmpty { };

struct FromCollection { Expr expr; Var as;      Var at; };
struct FromTuple      { Expr expr; Var as_name; Var as_value; };

struct FromInner { forward_ast<From> left; forward_ast<From> right; };
struct FromLeft  { forward_ast<From> left; forward_ast<From> right; };
struct FromFull  { forward_ast<From> left; forward_ast<From> right;
                   Expr cond; };
```

SQL++ sieht keine `SELECT`-Queries ohne `FROM`-Klausel vor. `FromEmpty` soll als Erweiterung des SQL++-Standards dienen und Queries der Form `SELECT ELEMENT 1` ermöglichen.

`FromCollection` repräsentiert `FROM`-Klauseln der Form `FROM <expr> AS <as> AT <at>`, bei welcher die Expression zu einem Bag oder Array evaluierbar sein muss. Der Wert jedes Elements in der Collection wird an die Variable `<as>` gebunden und der Index optional an `<at>`. Der `AT <at>` Teil der Klausel ist dabei optional und Fehlen wird im AST durch einen leeren `<at>` Variablennamen dargestellt.

`FromTuple` repräsentiert `FROM`-Klauseln der Form `FROM <expr> AS { <as_name> : <as_value> }`, bei welcher die Expression zu einem Tupel evaluierbar sein muss. Jeder Schlüssel im Tupel wird an die Variable `<as_name>` gebunden und der entsprechende Wert an `<as_value>`.

Die in SQL++ existierenden Joins basieren auf drei Variationen. Wie in Paragraph 2.2.5 beschrieben sind alle Join-Varianten als Syntaktischer-Zucker über `INNER|LEFT|FULL CORRELATE` definiert, teilweise durch Subqueries. Somit wird im AST nur `FromInner`, `FromLeft` und `FromFull` benötigt. Der Syntaktische-Zucker muss im Parser beachtet werden.

- `FromInner`: `INNER CORRELATE`, `INNER JOIN`, `INNER FLATTEN`
- `FromLeft`: `LEFT OUTER CORRELATE`, `RIGHT JOIN`, `LEFT JOIN`
- `FromFull`: `FULL OUTER CORRELATE`, `FULL JOIN`, `OUTER FLATTEN`

4.8.3 WHERE und HAVING

Die `WHERE`-Klausel (Darstellung 4.10) besteht aus einer einzigen Expression welche in der Ausführung zu True oder False evaluiert wird. Die `HAVING`-Klausel ist im AST identisch zur `WHERE`-Klausel, bezieht sich in der Ausführung jedoch auf Gruppen.

Darstellung 4.10 AST: WHERE/HAVING-Klausel

```
struct Where  { Expr expr; };
struct Having { Expr expr; };
```

4.8.4 GROUP BY

Die `GROUP BY`-Klausel (Darstellung 4.11) besteht aus einer Menge von Termen. Jeder Term besteht aus einer Expression sowie einem Variablennamen. Die Expression wird zur Gruppierung genutzt. Wenn der Variablenname nicht leer ist, wird der Wert der

Gruppierungs-Expression an diesen Namen gebunden. Die Grammatik erlaubt mehrere Gruppierungs-Expressions, weshalb `GroupBy` als Menge dieser Terme definiert ist.

Darstellung 4.11 AST: GROUP BY-Klausel

```
struct GroupBy_term {
  Expr expr;
  Var as;
};
struct GroupBy : vector<GroupBy_term> { };
```

4.8.5 ORDER BY

Die `ORDER BY`-Klausel (Darstellung 4.12) besteht aus einer geordneten Menge von Termen. Jeder Term besteht aus einer Expression sowie einem Boolean der beschreibt ob die Ordnung absteigend ist. Die Resultate werden nach den Werten der Expression der Terme sortiert, wobei die Reihenfolge der Terme beachtet wird.

Darstellung 4.12 AST: ORDER BY-Klausel

```
struct OrderBy_term {
  Expr expr;
  bool desc;
};
struct OrderBy : vector<OrderBy_term> { };
```

4.8.6 LIMIT und OFFSET

Die `LIMIT`- und `OFFSET`-Klauseln (Darstellung 4.13) bestehen je aus einer einzigen Expression, welche zu einer ganzen Zahl evaluierbar sein muss. Diese Zahl beschreibt die maximalen Elemente, beziehungsweise die Anzahl der ignorierten Elemente.

Darstellung 4.13 AST: LIMIT/OFFSET-Klausel

```
struct Limit  { Expr expr; };
struct Offset { Expr expr; };
```

Kapitel 5

Datentyp

Neben dem AST von Queries muss das SQL++-Datenmodell implementiert werden. *Cheesebase* besitzt bisher einen Datentyp der an JSON orientiert ist. Dieser muss erweitert werden um das SQL++-Datenmodell darstellen zu können.

Darstellung 5.1 Datenmodell Implementierung

```
struct Value : variant<Missing,
                       Null,
                       Number,
                       Bool,
                       String,
                       Shared<Tuple>,
                       Shared<Collection>> {};

struct Missing { };
struct Null { };
using Number = double;
using Bool = bool;
using String = string;
struct Tuple : map<String, Value> { };
struct Collection : vector<Value> { bool has_order; };
```

In Darstellung 5.1 ist eine vereinfachte Übersicht des neuen Datentypen dargestellt. `Value` ist eine Vereinigung über alle skalaren und komplexen Datentypen. `Missing` und `Null` besitzen je einen eigenen Typ, welcher leer ist. `Number`, `Bool` und `String` sind als neue Namen für Standardtypen implementiert. `Tuple` und `Collection` sind mit Hilfe von Containern der Standardbibliothek umgesetzt.

`Tuple` ist eine Abbildung von `String` auf `Value` und `Collection` eine Menge von `Value`s. `Collection` soll den Bag- und Array-Typ repräsentieren. Ob es sich um einen Bag oder Array handelt wird mittels der booleschen Variable `has_order` festgestellt. Diese

Kombination soll in der Ausführung erlauben einfach zwischen Bag und Array zu konvertieren. So kann in der Auswertung von `ORDER BY` ein gegebener Bag sortiert werden und die boolesche Variable auf Wahr gesetzt werden, womit das Kopieren in eine neue Collection umgangen wird.

Der im SQL++ enthaltene `enriched_value` wurde nicht implementiert. Da *Cheesebase* nur die ursprünglichen JSON-Typen implementiert ist der `enriched_value` nicht nötig.

In `Value` ist `Collection` und `Tupel` durch die `Shared` Hilfsklasse eingebunden. Diese enthält einen referenzgezählten Pointer auf die entsprechende `Collection` oder das `Tupel`. Wenn die `Shared`-Hilfsklasse kopiert wird findet keine Kopie des hinterlegten Wertes statt. Der Wert, in diesem Fall die `Collection` oder das `Tupel`, existieren einzigartig, wobei eine Vielzahl von `Shared`-Objekten auf diesen verweisen können. Sobald alle Referenzen auf einen Wert gelöscht wurden wird auch der Wert selbst gelöscht. Dies erlaubt das bewegen von `Value`s ohne große Datenmengen zu kopieren, da enthaltene komplexe Werte nur per Referenz gehalten werden.

In der Ausführung werden `Value`s auf verschiedenste Weise manipuliert und kombiniert, abhängig vom Inhalt der Query. Durch die Nutzung der `Shared`-Hilfsklasse werden dabei keine überflüssigen Kopien angefertigt. Alternativ wäre es möglich den gesamten `Value`-Typ als `Shared` zu implementieren. Jedoch kostet das Kopieren eines skalaren Wertes weniger Performance als das Anlegen und Nutzen eines `Shared`-Wertes. Aus diesem Grund wird diese Technik nur für die potential großen komplexen Typen genutzt. Jedoch ist zu beachten das während der Ausführung keine in-place Änderungen an `Shared`-Werten vorgenommen werden dürfen.

Kapitel 6

Parser

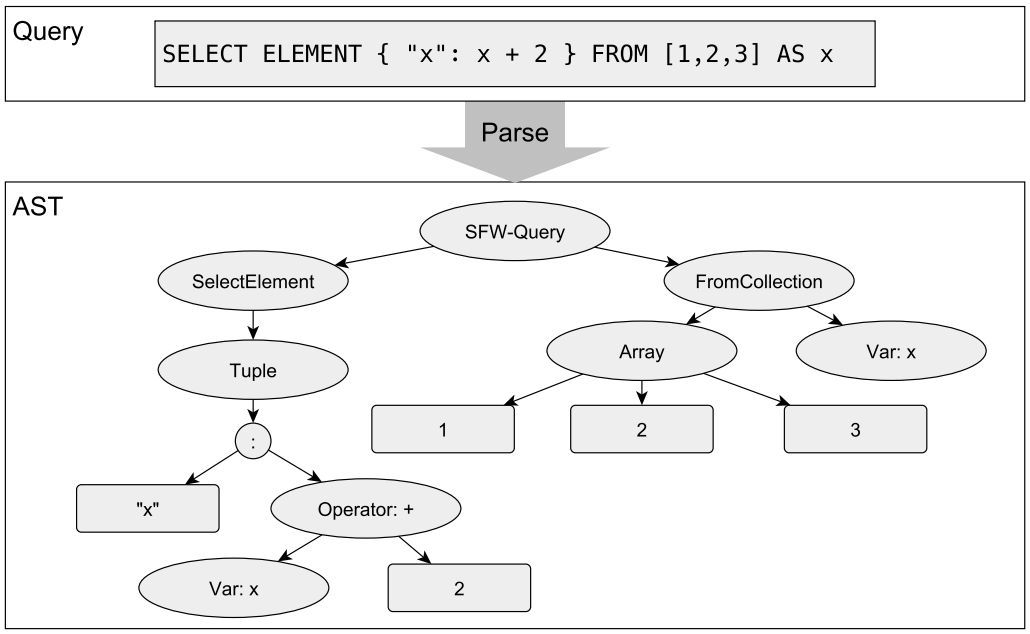


Abbildung 6.1: Query-Parsing

Beim Parsing soll aus der eingegebenen Query der AST generiert werden. Der AST wird anschließend zur Ausführung des Queries genutzt. In Abbildung 6.1 ist zu sehen wie der AST aus einer Beispielquery generiert wird, welcher anschließend zur Ausführung genutzt werden kann.

6.1 Grundlagen

6.1.1 Grammatik

Auf formaler Ebene besteht eine Grammatik aus vier Elementen [7]:

- Menge der **Nichtterminale**

  Symbole, oder Variablen, welche nicht im endgültigem Wort vorkommen. Werden in Zwischenschritten genutzt.

- Menge der **Terminale**

  Symbole welche im endgültigem Wort vorkommen. Ein endgültiges Wort besteht nur aus Terminalen.

- Menge der **Produktionen** oder **Regeln**

  Regeln besitzen die Form $l \rightarrow r$. l und r bestehen dabei aus einer Kombination von Terminal- und Nichtterminalsymbolen, wobei l mindestens ein Nichtterminal enthalten muss. Die Regeln beschreibt das l durch r ersetzt werden kann. Wenn die Regeln $l \rightarrow r_1$ und $l \rightarrow r_2$ existieren können diese zu $l \rightarrow r_1|r_2$ zusammengefasst werden.

- **Startvariable**

  Ein Nichtterminal welches der Ausgangspunkt der Grammatik ist.

Grammatiken können nach der *Chomsky-Hierarchie* in Typen eingeteilt werden [7]:

- **Typ 0**: Jede Grammatik, ohne Einschränkungen.
- **Typ 1**: Eine Grammatik ist *kontextsensitiv* wenn keine Regel existiert welche auf der linken Seite mehr Symbole vorkommen als auf der rechten Seite.
- **Typ 2**: Eine Typ 1 Grammatik ist *kontextfrei* wenn jede Regel auf der linken Seite nur aus einem einzelnen Nichtterminalsymbol besteht.
- **Typ 3**: Eine Typ 2 Grammatik ist *regulär* wenn jede Regel auf der rechten Seite nur aus Terminalen oder Terminalen gefolgt von einem Nichtterminal besteht.

Dies trifft auch auf die Grammatik von SQL++ zu, welche in Paragraph 2 vorgestellt wurde. Die SQL++-Grammatik ist kontextfrei (Typ 2).

Ein Beispiel einer einfachen kontextfreien Grammatik ist folgend dargestellt:

- **Nichtterminale**: { `<expr>` }
- **Terminale**: { `1`, `2`, `+` }

- **Produktionen**: { `<expr>` → `<expr> + <expr>` | `1` | `2` }
- **Startvariable**: `<expr>`

Der Ausdruck `1 + 2 + 3` kann, unter anderem, durch die Schritte welche in Abbildung 6.2 dargestellt sind abgeleitet werden. Der Ableitungsbaum ist für den Ausdruck nicht eindeutig, da mindestens ein weiterer valider Baum existiert. In diesen Fällen können Regeln festgelegt werden, wie immer das linke Nichtterminal zuerst abzuleiten und die erste valide Regel vorzuziehen. Durch diese Richtlinie beim Ableiten können auch nicht-eindeutige Grammatiken einen eindeutigen Ableitungsbaum generieren.

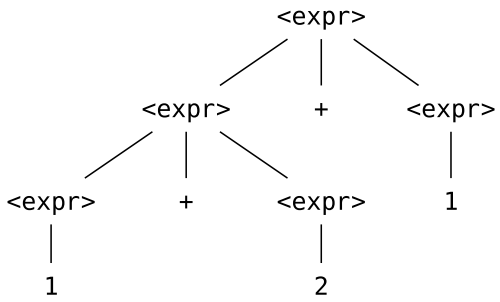


Abbildung 6.2: Ableitungsbaum des Beispiels

6.1.2 Lexikalische- und Syntaktische-Analyse

Das Parsen wird generell in zwei Schritte unterteilt. In der Lexikalischen-Analyse wird die eingegebene Zeichenkette in sogenannte Tokens unterteilt. Jedes Token entspricht einem Terminalsymbol der Grammatik.

In der Syntaktischen-Analyse werden die Token mit Hilfe der Produktionsregeln zu Ausdrücken kombiniert und der Syntax-Baum generiert.

Es existieren zwei grundlegende Techniken bei der Syntaktischen-Analyse, basierend auf der Reihenfolge in welche die Elemente im Syntax-Baum konstruiert werden. Top-Down-Parser beginnen an der Wurzel des Baumes und arbeiten sich in Richtung der Blätter vor. Bottom-Up-Parser generieren die Blätter des Baumes und kombinieren diese, bis eine einzige Wurzel erreicht wird. [1]

Eine typische Implementierung für einen Top-Down-Parser ist Recursive-Descent. Dabei existiert eine Prozedur für jedes Nichtterminal. Das Parsing beginnt mit der Prozedur für das Startsymbol. In einer Prozedur wird eine der Produktionsregeln für dieses Nichtterminal genutzt und die Prozeduren der dabei entstandenen Nichtterminale aufgerufen. Wenn eine Produktionsregel nicht erfüllt werden kann findet Backtracking

statt, womit eine andere Produktionsregel versucht werden kann. Durch die Möglichkeit von Backtracking ist wiederholtes Lesen der Eingabe möglich. Wenn das Backtracking keine Lösung finden kann oder die Ausgangsprozedur endet ohne die gesamte Eingabe konsumiert zu haben gab es einen Fehler.

Eine typische Implementierung für einen Bottom-Up-Parser ist Shift-Reduce. Dabei werden die Token der Eingabe nach und nach auf einen Stack geschoben. Mit jedem neuen Element wird versucht die oberen Elemente des Stacks anhand einer Produktionsregel zu kombinieren. Wenn eine Produktion angewandt wird werden die genutzten Elemente auf dem Stack durch das entstandene Nichtterminal ersetzt. Nach der Verarbeitung der gesamten Eingabe sollte auf dem Stack nur das Ausgangssymbol verbleiben.

6.2 Boost Spirit X3

Das *Boost Spirit* Framework [2] ist ein Parser-Generator für C++. Es generiert einen Recursive-Descent-Parser anhand von benutzerdefinierten Regeln. Die Regeln können in einer Backus-Naur-Form ähnlichen Syntax direkt im C++-Quellcode geschrieben werden. *Spirit* ist ein Teil der Boost Bibliothek, welche eine weit verbreitete und getestete C++ Bibliothek ist. *Boost* ist, motiviert durch andere Features, bereits in *Cheesebase* eingebunden.

Spirit stellt mit *Bison* und *Flex* die aktuell populärsten Parser-Frameworks dar, welche mit C++ kompatibel sind. *Bison* und *Flex* generieren C-basierte Parser mit Hilfe von externen Programmen. *Spirit* besitzt durch moderne C++ Template-Programmierung keine externen Abhängigkeiten. Aus diesem Grund, und da *Boost* bereits im Projekt eingebunden ist, ist die Entscheidung auf *Spirit* gefallen.

X3 ist die neuste Version des *Spirit* Frameworks und besitzt große Verbesserungen in Performance und Syntax.

6.2.1 Minimaler Parser

Darstellung 6.1 Spirit: Integer Parser

```
string input = "1234";

int result;
x3::parse(input.begin(), input.end(),
          x3::int_,
          result);
```

Darstellung 6.1 zeigt ein minimales Beispiel eines einfachen Parsers mit Spirit X3. `x3::int_` ist ein vorgefertigter Parser für Zahlen. Im Beispiel wird eine Variable für das Ergebnis angelegt (`result`) und anschließend der `x3::int_`-Parser auf die Zeichenkette `input` mit dem Wert `1234` angewandt. Der Parser wird durch die Funktion `x3::parse` ausgeführt. Nach Ausführung der `x3::parse` Methode ist die Variable `result` mit dem Wert `1234` belegt.

6.2.2 Zeichenketten Parser

Darstellung 6.2 Spirit: String Parser

```
string input = "Pizza";

string result;
x3::parse(input.begin(), input.end(),
          *x3::char_,
          result);
```

In Darstellung 6.2 ist das Parsen einer Zeichenkette gezeigt. Der Aufbau ist identisch zum letzten Beispiel. Das Ergebnis ist vom Typ Zeichenkette und die Eingabe hat den Wert `Pizza`. Der verwendete Parser ist `x3::char_`, welcher ein einzelnes Zeichen liest. Der Parser ist jedoch mit einem Operator versehen, einem vorangestellten Stern (`*`). Spirit stellt Operatoren bereit um Parser zu modifizieren. Der `*`-Operator ist identisch zum Kleene-Stern und bewirkt dass der Parser beliebig oft Ausgeführt werden kann (0 bis ∞ mal). Dabei verhält sich Spirit gierig und versucht möglichst viele Zeichen zu lesen. Der kombinierte Parser liefert somit eine Menge von Zeichen (`chars`). Spirit erkennt anschließend dass das Resultat vom Typ `string` ist und eine valide Konvertierung zwischen einer Menge von `chars` und einem `string` existiert. Somit hält die Variable `result` nach der Ausführung den Wert `Pizza`.

6.2.3 Kombinierter Parser

Darstellung 6.3 Spirit: kombinierter String Parser

```
string input = "Pizza Salad";

pair<string, string> result;
x3::parse(input.begin(), input.end(),
          *(~x3::char_(' ')) >> ' ' >> *x3::char_ ,
          result);
```

In Darstellung 6.3 wird eine Eingabe mit zwei Wörtern gelesen. Das Ergebnis ist ein Paar von zwei `string`s. Der kombinierte Parser nutzt verschiedene neue Operatoren.

Der >>-Operator verkettet zwei Parser und resultiert in einem neuen Parser der beide ursprüngliche Parser nacheinander ausführt. Der Resultattyp dieses neuen Parsers ist ein Paar der Ergebnisse beider Teilparser.

Auf der linken Seite wird der Parser `~x3::char_(' ')` verwendet. `x3::char_(' ')` ist ein Parser der nur das Zeichen `' '` (Leerzeichen) liest und zurück gibt. Der ~-Operator negiert diesen Parser, womit ein Parser erzeugt wird der alle Zeichen außer `' '` liest.

Dieser Parser wird mit dem zuvor vorgestellten *-Operator genutzt, womit alle Zeichen gelesen werden bis ein `' '` gefunden wird. Anschließend wird dieser Parser mit einem Leerzeichen-Literal verkettet (`... >> ' '`). Ein Literal prüft ob die nächsten Zeichen der Eingabe mit dem Wert des Literals übereinstimmen und liefert kein Ergebnis. Abschließend findet eine weitere Verkettung mit dem im letzten Beispiel genutzten Zeichenketten-Parser (`... >> *x3::char_`) statt.

Der endgültige Parser besteht aus drei verketteten Teilen: alle Zeichen bis zum ersten `' '`, lesen und ignorieren des Zeichens `' '`, und alle verbleibenden Zeichen. Da nur zwei der kombinierten Parser Ergebnisse liefern ist die Ausgabe ein Paar von zwei Mengen von `char`s. Spirit erkennt die mögliche Umwandlung auf den Typ `pair<string, string>` der Ausgabevariable. Somit hält die Variable `result` nach der Ausführung den Wert `("Pizza", "Salad")`.

6.2.4 Regeldefinition

Darstellung 6.4 Spirit: Regeldefinition

```
string input = "Pizza Salad Soup";

vector<string> result;
x3::rule<string> a_word = *(~x3::char_(' '));
x3::parse(input.begin(), input.end(),
          *a_word ,
          result);
```

In Darstellung 6.4 ist gezeigt wie eine neue Regel definiert wird. Der erste Parser aus dem letzten Beispiel wird hier als `a_word` gespeichert. In der Regel ist der Resultat-Typ explizit definiert (`string`). Im Aufruf des Parsers wird der *-Operator auf die definierte Regel angewandt, was in einer Menge von `string`s resultiert. Die `result` Variable ist vom Typ `vector<string>`, welcher erfolgreich aus einer Menge von `string`s generiert werden kann, womit der Wert der Variable `{ "Pizza", "Salad", "Soup" }` ist.

6.2.5 Alternativen

In Darstellung 6.5 ist gezeigt wie Alternativen umgesetzt werden können. Der Typ der `result` Variable ist ein `vector` von `variant<int, string>`, ein `variant` kann einen Wert von einem der beiden Typen halten. Im Aufruf wird der `|`-Operator genutzt um den `x3::int_`-Parser mit dem zuvor definierten `a_word`-Parser zu kombinieren. Der dabei resultierende Parser versucht erst eine Zahl zu lesen, wenn dies fehlschlägt wird versucht ein Wort zu lesen.

Darstellung 6.5 Spirit: Alternativen

```
string input = "Pizza 15 Soup 3";

vector< variant<int, string> > result;
x3::rule<string> a_word = *(~x3::char_(' '));
x3::parse(input.begin(), input.end(),
          *( x3::int_ | a_word ) ,
          result);
```

Der Resultattyp eines Parsers der mit diesem Operator generiert wurde ist `variant<Typ_links, Typ_rechts>`. Wobei Spirit mehrere `variant` Kombinationen vereinen kann. So würde die Kombination von Parsern mit den Resultattypen `A`, `B` und `C` ein `variant<A, B, C>` ergeben. Wenn der Aufruf `a_word | x3::int_` wäre (Parser getauscht), würde immer ein `string` gelesen werden, da die Parser von links nach rechts angewandt werden und der `a_word`-Parser auch Zahlen als Wort akzeptiert. Der Wert von `result` nach der Ausführung ist `{ "Pizza", 15, "Soup", 3 }`, wobei die Zahlen Integer sind und keine Zeichenkette von Ziffern.

6.2.6 Semantic-Actions

In Darstellung 6.6 ist gezeigt wie Semantic-Actions genutzt werden können. Dabei handelt es sich um Funktionen welche ausgeführt werden wenn ein Parser erfolgreich angewandt wurde. Der Funktion wird ein Kontext des Parsers übergeben der es erlaubt auf den gelesenen Wert zuzugreifen oder den Rückgabewert des Parsers zu manipulieren. Im Beispiel ist `add` eine Lambda-Funktion die als Semantic-Action für den Parser `x3::int_` genutzt wird. Die `add` Funktion extrahiert den gelesenen Wert des Parsers mit `x3::_attr(ctx)` und addiert diesen zur Variable `sum`.

Der Parser wird durch den `%`-Operator wiederholt angewandt. Der `%`-Operator wird genutzt um Listen mit einem Separator zu parsen. Im Beispiel wird eine Zahl verarbeitet, wenn auf diese ein `','` folgt, wird eine weitere Zahl verarbeitet. Leere Zeichen werden bei diesen Verbindungen generell ignoriert. Somit verarbeitet der kombinierte

Darstellung 6.6 Spirit: Semantic-Actions

```
string input = "1, 2, 3, 4, 5";

int sum = 0;
auto add = [&sum](auto& ctx) { sum += x3::_attr(ctx); };

x3::parse(input.begin(), input.end(),
          ( x3::int_[add] ) % ','
         );
```

Parser die Zeichenkette `"1, 2, 3, 4, 5"` vollständig. Der Wert von `sum` ist 15 nach erfolgreichem Ausführen des Parsers.

Semantic-Actions können unter anderem genutzt werden um komplexe Rückgabewerte zu konstruieren.

6.2.7 Zusammenfassung

Die vorgestellten Features stellen nur einen kleinen Teil der Fähigkeiten von *Boost Spirit X3* dar. Es existieren eine Vielzahl von weiteren Operatoren und vorgefertigten Parsern. Durch das Definieren von neuen Regeln lassen sich schnell komplexe Grammatiken konstruieren. Mit einfachen Kombinieren von Rückgabewerten können die meisten Knoten im AST generiert werden. Wenn dies nicht ausreicht ist mit Semantic-Actions ein mächtiges Werkzeug gegeben um komplexe Strukturen zu erzeugen.

6.3 Expression

Der Kern der SQL++-Grammatik ist die Expression, welche auch den Ausgangspunkt des Parsers darstellt. In Darstellung 6.7 ist die Definition des `expr` und `query`-Parsers dargestellt. Beide Parser liefern ein `Expression`-Objekt aus dem Query-AST.

Darstellung 6.7 Parser: Expression

```
x3::rule<Expression> expr = '(' >> sfw_query >> ')'
                          | function
                          | tuple
                          | array
                          | bag
                          | scalar
                          | var;

x3::rule<Expression> query = sfw_query | expr;
```

Der `expr`-Parser kombiniert eine Vielzahl von Parsern welche im folgenden definiert werden. Jeder dieser Parser gibt ein Objekt vom entsprechenden Typ aus dem Query-AST zurück. Durch den `|`-Operator wird versucht alle Parser zu nutzen bis der erste erfolgreich angewandt werden konnte. Das resultierende Objekt wird genutzt um das `Expression`-Objekt zu generieren, was möglich ist da `Expression` als `variant` über alle Untertypen definiert wurde.

Der `query`-Parser stellt den eigentlichen Ausgangspunkt dar. Dieser spiegelt wieder dass es sich bei einer Query um eine SFW-Query oder Expression handelt, wobei eine Expression eine in Klammern gefasste Subquery enthalten kann.

Die Definition enthält keine Navigationen oder Algebra-Operationen. Da diese, wie später erläutert, über Umwege definiert werden müssen.

6.4 Werte

Eine grundlegende Funktion eines SQL++-Query-Parsers ist es SQL++-Values zu lesen. In Queries können dabei nur skalare Werte vorkommen, da komplexe Werte in der Query-Grammatik als Konstruktoren bestehend aus einer Menge von Expressions definiert sind.

Darstellung 6.8 Parser: Value

```
x3::rule<Null>       null = "null" >> x3::attr(Null());
x3::rule<Missing> missing = "missing" >> x3::attr(Missing());
x3::rule<Number>   number = x3::double_;
x3::rule<Bool>   bool_val = "true" >> x3::attr(true)
                          | "false" >> x3::attr(false);
x3::rule<String>   string = x3::lexeme['"' >> *(~x3::char_('"')) >> '"'];

x3::rule<Value>    scalar = missing | null | number | bool_val | string;
```

Darstellung 6.8 zeigt die Definitionen der Parser für skalare Werte. Im Parser `null`, `missing` und `bool_val` wird `x3::attr(...)` genutzt, dabei handelt es sich um einen Dummy-Parser der das im Argument gegebene Objekt als Rückgabewert liefert. Dieser Dummy-Parser wird mit einem Literal-Parser (`"null"`, `"missing"`, `"true"`, `"false"`) kombiniert, wenn das entsprechende Literal erfolgreich gelesen wurde wird der Dummy-Parser ausgeführt und liefert das entsprechende Objekt.

Der `string`-Parser liest eine mit Anführungszeichen umgebene Zeichenkette. `x3::lexeme[...]` unterdrückt das Überspringen von leeren Zeichen. Der `string`-Parser ist hierbei nur naiv implementiert. Speziell codierte Zeichen (nicht ASCII) werden nicht verarbeitet und Maskierungszeichen (`" \" "`) sind nicht implementiert.

Die Parser werden im `scalar`-Parser kombiniert, welcher ein `Value`-Objekt mit dem entsprechenden Wert zurück gibt.

6.5 Variablen

In Darstellung 6.9 wird eine `name`-Parser eingeführt. Dieser liest Zeichenketten welche nur aus Buchstaben, Ziffern und dem Symbol `'_'` bestehen und mit einem Buchstaben beginnen. Dieser Parser gibt einen einfachen `string` zurück und wird für Variablen und Funktionsnamen genutzt. Der `var`-Parser wendet den `name`-Parser an und konvertiert das Ergebnis zum Typ `Var` aus dem Query-AST.

Darstellung 6.9 Parser: Variable

```
x3::rule<string> name = x3::lexeme[ x3::alpha >>
                                    *(x3::alnum | x3::char_('_')) ];
x3::rule<Var> var = name;
```

6.6 Funktionen

In Darstellung 6.10 wird der `function`-Parser definiert der einen Funktionsaufruf liest. Ein Funktionsaufruf besteht aus dem Funktionsnamen sowie einer liste von Argumenten. Der `expr % ','` Teil liest eine Expression und anschließend mit Komma getrennte weitere Expressions. Der `expr`-Parser wird später definiert. Der '`-`'-Operator, welcher auf den `expr % ','` Teil angewandt wird, signalisiert das dieser Teil optional ist. Somit wird auch eine leere Liste, also eine Funktion ohne Argumente, akzeptiert.

Darstellung 6.10 Parser: Funktion

```
x3::rule<Function> function = name >> '(' >> -(expr % ',') >> ')';
```

Der Rückgabewert des gesamten Parsers kann auf einen `string` gefolgt von einer Menge von Expressions reduziert werden. Diese Struktur entspricht der Struktur des `Function`-Typs des Query-AST, womit ein `Function`-Objekt konstruiert werden kann.

6.7 Komplexe Werte

Darstellung 6.11 zeigt die Konstruktoren für komplexe Werte.

Der `tupel_member`-Parser gibt ein Paar aus einer Zeichenkette und einer Expression zurück. Die Zeichenkette wird vom zuvor definierten `name` oder `string`-Parser gelesen.

Darstellung 6.11 Parser: Sammlungen

```
x3::rule<pair<string, Expr>> tuple_member = (name | string) >> ':' >> expr;
x3::rule<Tuple> tuple = '{'  >> -(tuple_member % ',') >> '}';
x3::rule<Array> array = '['  >> -(expr % ',')         >> ']';
x3::rule<Bag>     bag = "{{" >> -(expr % ',')         >> "}}";
```

Diesem folgt das lesen eines Literals (`':'`) sowie einer Expression. Die alternative Nutzung des `name` und `string`-Parser erlaubt das Lesen von Tupeln der Form `{ simple_name : 1, "komplex name" : 2 }`. Die Nutzung von Anführungszeichen bei einfachen Namen ist somit optional.

Im `tuple`-Parser wird der `tupel_member`-Parser wiederholt angewandt. Dies geschieht, wie bei dem `function`-Parser, durch eine Kombination von `%`- und '-'-Operator. Der Rückgabewert des kombinierten Parsers ist eine Menge aus String-Expression-Paaren, welche für die Konstruktion des `Tuple`-Typs aus dem Query-AST genutzt werden können.

Der `array` und `bag`-Parser liest je eine durch Kommas getrennte Liste von Expressions. Die Parser unterscheiden sich nur in den umgebenden Literalen und dem Rückgabewert.

6.8 Pfad-Navigationen

Array- und Tupel-Navigation sind als Expression gefolgt von einem Argument definiert. Das Argument ist bei einer Tupel-Navigation mit einem Punkt abgetrennt (`tuple.subelement`) und bei einer Array-Navigation mit Klammern umgeben (`array[index]`). Die Navigation bildet selbst eine Expression und ist somit rekursiv definiert.

Speziell handelt es sich um eine Linksrekursion, da sich das entsprechende Nichtterminal links befindet. Top-Down-Parser können Linksrekursionen nicht verarbeiten.

Darstellung 6.12 Parser: naive Tupel-Navigation

```
x3::rule<Expression>    expr = tuple_nav | ... ;
x3::rule<TupleNav> tuple_nav = expr >> '.' >> name;
```

Sollte die Tupel-Navigation wie in Darstellung 6.12 implementiert werden endet das Parsing in einer unendlichen Rekursion. Wenn eine Zeichenkette mit dem `expr`-Parser interpretiert wird versucht dieser den `tupel_nav`-Parser anzuwenden. Der `tupel_nav`-Parser versucht anschließend den `expr`-Parser anzuwenden welcher erneut den `tupel_nav`-Parser nutzt, womit der Parser nie terminiert.

Dieses Problem kann bei der Benutzung von Top-Down-Parsern nur durch Anpassung der Grammatik umgangen werden. Es existieren Transformationen zur Vermeidung von Linksrekursionen. [4]

Mit den linksrekursiven Produktionen

$$A \to A\alpha_1|...|A\alpha_n$$

und den verbleibenden Produktionen

$$A \to \beta_1|...|\beta_m$$

können alle Produktionen durch

$$A \to \beta_1|\beta_1 A'|...|\beta_m|\beta_m A'$$

und

$$A' \to \alpha_1|\alpha_1 A'|...|\alpha_n|\alpha_m A'$$

ersetzt werden.

Durch Anwenden dieser Transformation auf das Beispiel aus Darstellung 6.12 und weiterer Vereinfachung kann der Parser aus Darstellung 6.13 definiert werden.

Darstellung 6.13 Parser: angepasste Tupel-Navigation

```
x3::rule<Expression>  expr0 = ... ;
x3::rule<???>          expr = expr0 >> tuple_nav | expr0;
x3::rule<???>     tuple_nav = '.' >> name >> -tuple_nav;
```

Diese Definition besitzt jedoch zwei Probleme. Zum einen sind die Rückgabewerte der Parser nicht klar und es wird unnötiges mehrfaches Parsing durchgeführt. Beim lesen einer Expression wird eine Basisexpression (`expr0`) verarbeitet und anschließend versucht eine Tupel-Navigation zu lesen. Wenn keine Tupel-Navigation gefunden wurde wird die Basisexpression verworfen und erneut, ohne folgende Tupel-Navigation, gelesen.

Um den Parser zu vereinfachen kann die optionale Rechtsrekursion in `tupel_nav` durch Nutzung der positiven Hülle umgangen werden. Zusätzlich wurde der Inhalt des `tuple_nav`-Parsers direkt in den `expr`-Parser eingebunden.

```
x3::rule<???>  expr = expr0 >> +('.' >> name) | expr0;
```

Der +-Operator entspricht der positiven Hülle (1 bis n mal). Mit dem Kleene-Stern kann der Parser weiter vereinfacht werden.

```
x3::rule<???>  expr = expr0 >> *('.' >> name);
```

Dieser Parser kann Tupel-Navigationen effizient lesen. Jedoch ist der Rückgabewert des kombinierten Parsers nicht direkt zum generieren der AST-Knoten geeignet. Mit Semantic-Actions kann der Rückgabewert manuell generiert und angepasst werden.

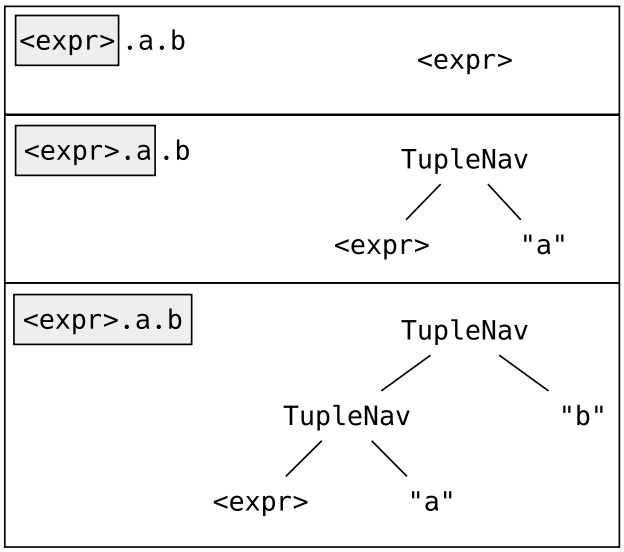


Abbildung 6.3: TupleNav-Parser

Wenn der mit dem Kleene-Stern versehene rechte Teil des Parsers nie angewandt wird entspricht der Rückgabewert der Expression welche von `expr0` gelesen wird. Bei jedem Anwenden des rechten Parsers ändert sich der aktuelle Rückgabewert zu einem `TupleNav`-Objekt welches den alten Rückgabewert sowie den entsprechenden Namen hält. Dieser Vorgang ist in Abbildung 6.3 dargestellt.

In der finalen Implementation (Darstellung 6.14) wird die Array- und Tupel-Navigation im gleichen Parser implementiert. Als Semantic-Actions werden die Funktionen `assign`- und `rotate_2` genutzt. `assign` setzt den Rückgabewert auf den resultierenden Wert des aufgerufenen Parsers. `rotate_2` generiert ein neues `TupleNav`- oder `ArrayNav`-Objekt und rotiert den zuvor gesetzten Rückgabewert in das neue Objekt.

Darstellung 6.14 Parser: finale Tupel-Navigation

```
auto assign = [](auto& ctx) {
  x3::_val(ctx) = x3::_attr(ctx);
};

template <typename T>
auto rotate_2 = [](auto& ctx) {
  x3::_val(ctx) = T(x3::_val(ctx), x3::_attr(ctx));
};

x3::rule<Expression> expr = expr0[assign]
        >> *( '.' >> name[rotate_2<TupleNav>]
            | '[' >> expr[rotate_2<ArrayNav>] >> ']'
            );
```

6.9 Operatoren

Die Infix-Operatoren sind ebenfalls linksrekursiv definiert. Zusätzlich muss der Operatoren-Vorrang beachtet werden. Ähnlich wie bei den Navigationen werden auch die Operatoren implementiert indem neue Ebenen von Expressions eingeführt werden.

```
x3::rule<???> expr2 = expr1 >> -(infix_op >> expr1);
```

Somit wird Linksrekursion verhindert und durch mehrere Ebenen kann Operatoren-Vorrang implementiert werden. Um mehrere Operatoren mit gleichem Vorrang zu verarbeiten wird erneut der Kleene-Stern verwendet.

```
x3::rule<???> expr2 = expr1 >> *(infix_op >> expr1);
```

In jeder Ebene werden Operatoren mit identischen Vorrang verarbeitet. Die Parser für alle aktuell implementierten Operatoren nach Ebene sind in Darstellung 6.15 dargestellt. Der `x3::attr`-Dummy-Parser wird verwendet um bei erfolgreich gelesenem Operatoren-Literal das entsprechende Operatoren-Objekt aus dem Query-AST zurück zu geben.

Alle implementierten Ebenen der Expressions sind in Darstellung 6.16 dargestellt. Niedrige Ebenen besitzen die stärkste Operatoren-Bindung.

Der Parser für Navigationen wurde zu `expr2` umbenannt und bezieht sich auf `expr1`. `expr1` wurde eingeführt um geklammerte Ausdrücke zu verarbeiten. Darauf folgt der Prefix-Operator, welcher nicht als Linksrekursion definiert ist und somit simpel implementiert werden kann. Anschließend sind die Parser für alle Infix-Operatoren in vier Ebenen aufgeteilt. Diese ähneln der Implementierung der Navigationen, nutzen jedoch `rotate_3` als Semantic-Action. Die `rotate_3`-Funktion generiert Knoten aus drei Ele-

Darstellung 6.15 Parser: Operatoren

```
x3::rule<PrefixOperator> prefix1_def =
        '-' >> x3::attr(PrefixOperator::neg);
x3::rule<InfixOperator> infix1_def =
        '*'  >> x3::attr(InfixOperator::mul)
      | '/'  >> x3::attr(InfixOperator::div)
      | '%'  >> x3::attr(InfixOperator::modulo);
x3::rule<InfixOperator> infix2_def =
        '+'  >> x3::attr(InfixOperator::plus)
      | '-'  >> x3::attr(InfixOperator::minus);
x3::rule<InfixOperator> infix3_def =
        "<=" >> x3::attr(InfixOperator::le)
      | '<'  >> x3::attr(InfixOperator::lt)
      | ">=" >> x3::attr(InfixOperator::ge)
      | '>'  >> x3::attr(InfixOperator::gt)
x3::rule<InfixOperator> infix4_def =
        "==" >> x3::attr(InfixOperator::eq)
      | '='  >> x3::attr(InfixOperator::eq)
      | "!=" >> x3::attr(InfixOperator::neq)
      | "<>" >> x3::attr(InfixOperator::neq);
```

menten. Die Funktion erhält einen Operator sowie eine Expression als Argument und generiert einen `InfixOp`-Knoten welcher den Operator, die ursprüngliche Expression als linke und die neue Expression als rechte Seite enthält.

In Abbildung 6.4 sind die bei der Verarbeitung der Expression

```
-(5 + 9 * 3) == x[10]
```

genutzten Parser visualisiert. Der daraus resultierende AST ist in Abbildung 6.5 dargestellt.

6.10 SFW-Query

Der Parser für SELECT-FROM-WHERE-Queries besteht aus einer Verkettung der einzelnen Klauseln. Die SELECT-Klausel muss existieren. Wenn die FROM-Klausel existiert kann eine beliebige Kombination weiterer Klauseln folgen. Ohne FROM-Klausel besteht die Query nur aus der SELECT-Klausel. In Darstellung 6.17 ist der Parser dargestellt. Alle auf den `select`-Parser folgenden Klauseln sind in einem optionalen Block zusammen gefasst. Innerhalb des Blocks befindet sich der `from`-Parser und alle weiteren optionalen Parser. Die Struktur der Parser entspricht der Struktur des AST-Knotens. Da im AST alle nicht garantierten Klauseln als `optional` implementiert sind kann *Spirit* erfolgreich ein `SfwQuery`-Objekt generieren.

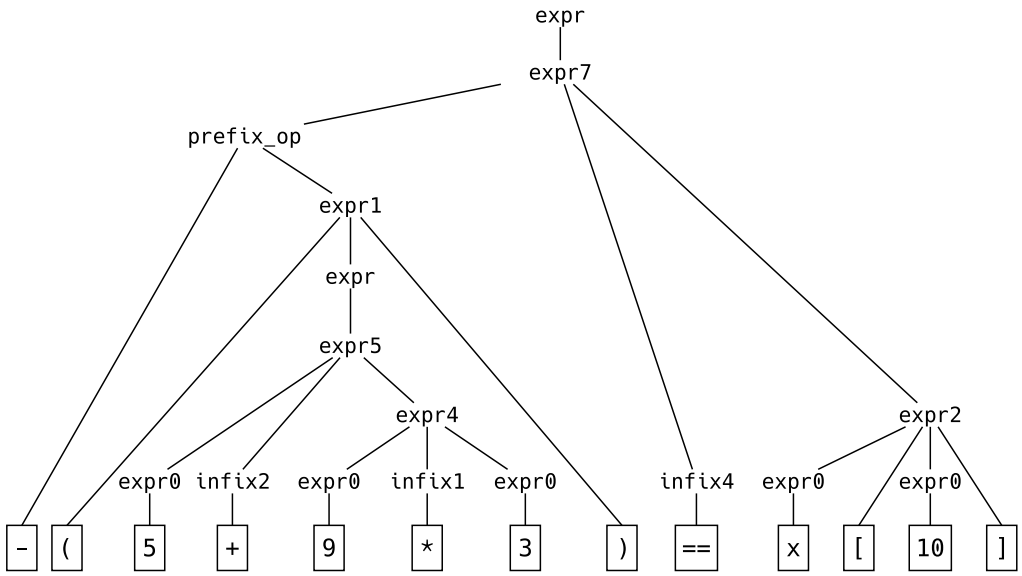


Abbildung 6.4: Beispiel Parsetree

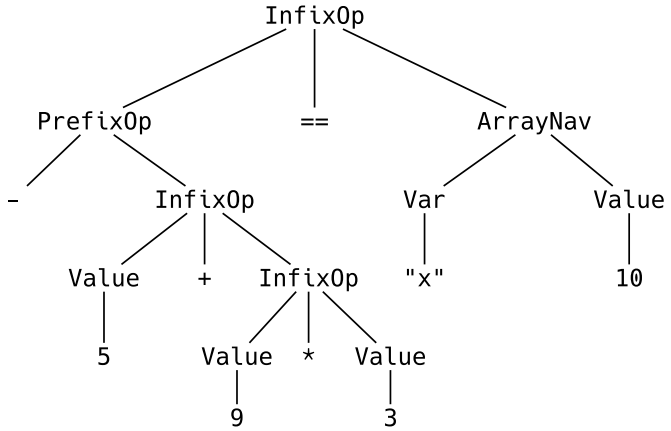


Abbildung 6.5: Beispiel AST

Darstellung 6.16 Parser: Expression-Ebenen

```
x3::rule<Expression> expr0 = ... // Basis
x3::rule<Expression> expr1 = expr0 | '(' >> expr >> ')';
x3::rule<Expression> expr2 = ... // Navigationen

x3::rule<PrefixOp> prefix_op = prefix1 >> expr;
x3::rule<Expression> expr3 = prefix_op | expr2;

x3::rule<Expression> expr4 = expr3[assign]
        >> *( (infix1 >> expr3)[rotate_3<InfixOp>] );
x3::rule<Expression> expr5 = expr4[assign]
        >> *( (infix2 >> expr4)[rotate_3<InfixOp>] );
x3::rule<Expression> expr6 = expr5[assign]
        >> *( (infix3 >> expr5)[rotate_3<InfixOp>] );
x3::rule<Expression> expr7 = expr6[assign]
        >> *( (infix4 >> expr6)[rotate_3<InfixOp>] );

x3::rule<Expression> expr = expr7;
```

Darstellung 6.17 Parser: SFW-Query

```
x3::rule<SfwQuery> sfw_query = select >> -(from
                                        >> -where
                                        >> -group_by
                                        >> -order_by
                                        >> -limit
                                        >> -offset);
```

6.10.1 SELECT

Die SELECT-Klausel (Darstellung 6.18) besitzt drei Varianten. `SELECT ATTRIBUTE` besteht aus zwei Expressions welche mit einem `:` getrennt sind und kann direkt generiert werden (`select_attribute`-Parser). `SELECT ELEMENT` besteht aus einer einfachen Expression und kann ebenfalls direkt generiert werden (erste Alternative im `select_element`-Parser).

Darstellung 6.18 Parser: SELECT-Klausel

```
x3::rule<Select> select = "SELECT" >> (select_attribute | select_element);
x3::rule<SelectAttribute> select_attribute =
                          "ATTRIBUTE" >> expr >> ':' >> expr;
x3::rule<SelectElement> select_element =
                          ("ELEMENT" >> expr) | select_element_list;
```

Zusätzlich existiert das SQL artige `SELECT x AS a, y AS b, ...` als syntaktischer Zucker für `SELECT ELEMENT { a:x, b:y, ...}`. `select_element_list_item` (Darstellung 6.19) liest ein Paar aus dieser Liste. Dabei wird die Expression gelesen und durch eine Semantic-Action als zweites Element im zurück gegebenen Paar gesetzt. Der

darauf folgende Name ist optional. Wenn er existiert wird das erste Element im Paar auf diesen Namen gesetzt. Sollte kein Name gegeben sein wird versucht die Expression als `Var` zu interpretieren und den Variablennamen als Namen im Paar zu verwenden, sollte dies nicht möglich sein wird ein Fehler ausgelöst. `select_element_list` liest eine Komma getrennte Liste dieser Paare und generiert ein `Tuple` welches als Expression im `SelectElement`-Objekt genutzt wird.

Darstellung 6.19 Parser: SELECT-Liste

```
x3::rule<Tuple> select_element_list = select_element_list_item % ',';

x3::rule<pair<string, Expr>> select_element_list_item =
  expr[ [](auto& ctx) { x3::_val(ctx).second = x3::_attr(ctx); } ]
  >> ( "AS" >> name[ [](auto& ctx) {
         x3::_val(ctx).first = x3::_attr(ctx);
       } ]
     |
       x3::eps[ [](auto& ctx) {
         auto name = get<Var>(&x3::_val(ctx).second);
         if (name == nullptr)
           throw ParserError{ "Could not derive name for SELECT pair" };
         x3::_val(ctx).first = *name;
       } ]
     );
```

6.10.2 FROM

Die FROM-Klausel beinhaltet alle Join-Varianten. Wie in Darstellung 6.20 zu sehen liest der Parser das Literal `"FROM"` gefolgt von einem `from_item`. Der `from_item`-Parser ist rekursiv definiert und ist im Aufbau ähnlich dem zuvor definiertem Operatoren-Parser.

Darstellung 6.20 Parser: FROM-Klausel

```
x3::rule<From> from = "FROM" >> from_item;
```

Die Basis des `from_item`-Parsers bilden `from_collection` und `from_tuple`. Beide können direkt einen AST-Knoten generieren. Beide werden im Parser `from_item0` kombiniert welcher die unterste Ebene entspricht. In dieser Ebene sind zusätzlich Klammern implementiert womit verschiedenste Join-Strukturen umgesetzt werden können.

Die oberen Ebenen lesen die im SQL++ direkt enthaltenden Join-Varianten (`INNER|LEFT|FULL CORRELATE`). Wobei `FULL JOIN` und `FULL CORRELATE` identisch sind. Die Parser sind äquivalent zu dem Parser der Infix-Operatoren implementiert. Es wird ein `from_item` der nächst niederen Ebene gelesen und als Rückgabewert gesetzt.

Darstellung 6.21 Parser: Basis FROM-Items

```
x3::rule<FromCollection> from_collection =
                  expr >> "AS" >> var >> -("AT" >> var);
x3::rule<FromTuple> from_tuple =
                  expr >> "AS" >> '{' >> var >> ':' >> var >> '}';
x3::rule<From> from_item0 =
                  from_collection | from_tuple | '(' >> from_item >> ')';
```

Anschließend wird nach dem kombinierendem Literal gesucht, wenn dieses gefunden wurde wird ein weiteres `from_item` gelesen und der Rückgabewert mittels `rotate_x` manipuliert. Dieser Schritt wird mittels Kleene-Stern wiederholt ausgeführt. Im Parser für `FULL CORRELATE` wird außerdem die Expression `ON expr` gelesen.

Darstellung 6.22 Parser: INNER|LEFT|FULL CORRELATE

```
x3::rule<From> from_item4 = from_item3[assign] >>
                            *(("INNER" >> -"CORRELATE" >>
                               from_item3)[rotate_2<FromInner>]);
x3::rule<From> from_item5 = from_item4[assign] >>
                            *(("LEFT" >> -"OUTER" >> -"CORRELATE" >>
                               from_item4)[rotate_2<FromLeft>]);
x3::rule<From> from_item6 = from_item5[assign] >>
                            *(("FULL" >> ("JOIN"
                                          | (-"OUTER" >> -"CORRELATE"))
                            >> from_item5
                            >> "ON" >> expr)[rotate_3<FromFull>]);
```

Die weiteren Ebenen implementieren die Varianten welche als syntaktischer Zucker definiert sind. Wie in Darstellung 6.23 dargestellt sind die Parser ähnlich wie die vorherigen implementiert. Statt einfacher Konstruktion der AST-Knoten mittels `rotate_x` werden manuell Queries generiert. In `from_item1` wird aus

```
<left> INNER JOIN <r> AS <v> ON <c>
```

ein Knoten entsprechend des FROM-Items

```
<left> INNER CORRELATE
    (SELECT ELEMENT <v> FROM <r> AS <v> WHERE <c>) AS <v>
```

generiert.

Die nicht dargestellten Parser `from_item2` und `from_item3` implementieren entsprechende Transformationen für `LEFT JOIN` und `RIGHT JOIN`.

Darstellung 6.23 Parser: INNER JOIN

```
x3::rule<From> from_item1 = from_item0[assign]
  >> *(("INNER JOIN" >> from_collection
        >> "ON" >> expr)[ [](auto& ctx) {
    auto& from = at_c<0>(x3::_attr(ctx));
    FromCollection right;
    right.as = from.as;
    SfwQuery sfw;
    sfw.select = SelectElement(from.as);
    sfw.from = from;
    sfw.where = at_c<1>(x3::_attr(ctx));
    right.expr = sfw;
    x3::_val(ctx) = FromInner(x3::_val(ctx), right);
  } ]);
```

6.10.3 WHERE

Der WHERE-Parser (Darstellung 6.24) liest eine einzelne Expression. Die Expression muss in der Ausführung in einem booleschen Wert resultieren, was beim Parsing nicht geprüft werden kann.

Darstellung 6.24 Parser: WHERE

```
x3::rule<Where> where = "WHERE" >> expr;
```

6.10.4 ORDER BY

Der ORDER-BY-Parser (Darstellung 6.25) liest eine Komma getrennte Liste von Termen ein. Jeder Term besteht aus einer Expression sowie einem optional folgendem `ASC` oder `DESC`. Die AST-Klasse `OrderBy_term` wurde angepasst um die Variable für die Sortierrichtung (`desc`) automatisch auf Aufsteigend (`false`) zu setzen. Nur wenn der Literal `DESC` gefunden wurde wird mit einer Semantic-Action der Wert dieser Variable auf Absteigend (`true`) gesetzt.

Darstellung 6.25 Parser: ORDER BY

```
x3::rule<OrderBy_term> order_by_term =
  expr[ [](auto& ctx) { x3::_val(ctx).expr = x3::_attr(ctx); } ] >>
  -("ASC" | "DESC")[ [](auto& ctx) { x3::_val(ctx).desc = true; } ];

x3::rule<OrderBy> order_by = "ORDER BY" >> (order_by_term % ',');
```

6.10.5 GROUP BY

Der GROUP-BY-Parser (Darstellung 6.26) liest eine Komma getrennte Liste von Termen ein. Jeder Term besteht aus einer Expression sowie einer optionalen Variable. Wenn die Variable gegeben ist wird dieser in der Ausführung der Wert der Gruppierungs-Expression zugewiesen.

Darstellung 6.26 Parser: GROUP BY

```
x3::rule<GroupBy_term> group_by_term = expr >> -("AS" > var);
x3::rule<GroupBy> group_by = "GROUP BY" >> (group_by_term % ',');
```

6.10.6 LIMIT und OFFSET

Der LIMIT- und OFFSET-Parser (Darstellung 6.27) besteht je aus dem Literal gefolgt von einer Expression. Die Expression muss in der Ausführung eine natürliche Zahl ergeben.

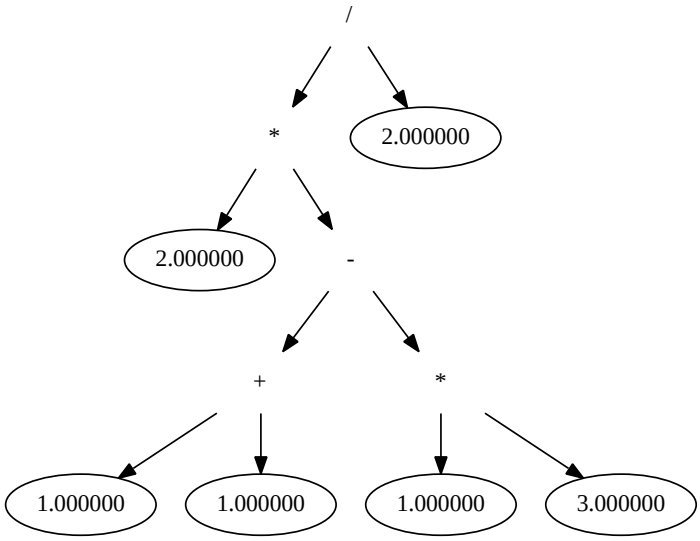
Darstellung 6.27 Parser: LIMIT/OFFSET

```
x3::rule<Limit> limit = "LIMIT" >> expr;
x3::rule<Offset> offset = "OFFSET" >> expr;
```

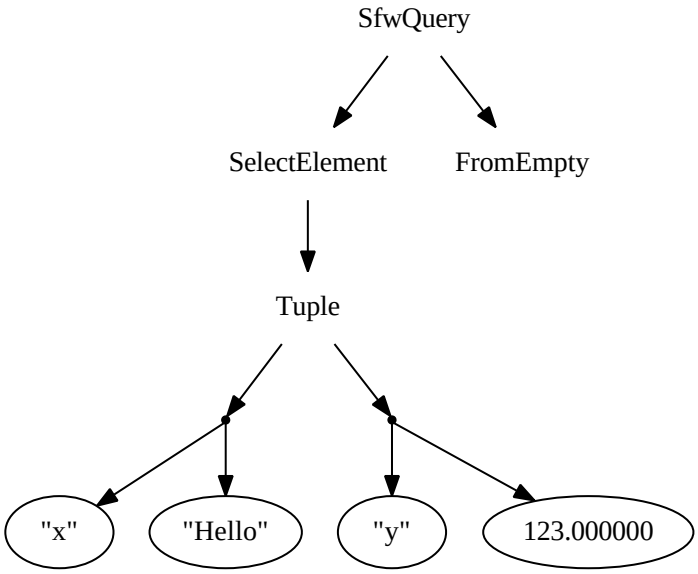
6.11 Beispiele

Nach Implementierung aller Teilparser kann aus Queries der AST generiert werden. Um die korrekte Funktionalität zu prüfen wurde ein System implementiert um den AST als Graph auszugeben. Dabei wird der AST durchlaufen und Text im *DOT*-Format ausgegeben. *DOT* ist eine einfache Beschreibungssprache für die visuelle Darstellung von Graphen. Mit dem gleichnamigen Tool *dot* wird aus diesem Text ein Bild generiert. Innere AST-Knoten werden im Graph als einfacher Text dargestellt, Variablen in einem Kasten und Werte in einem Oval.

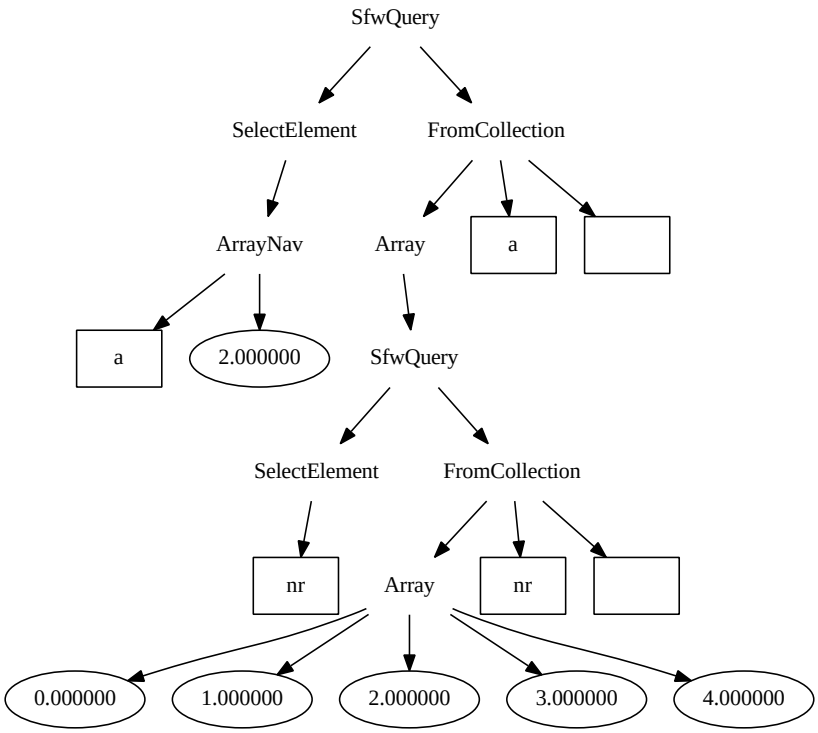
- **`(2 * (1 + 1 - 1 * 3)) / 2`**
  Kombination verschiedener mathematischer Operatoren. Es ist zu erkennen dass der Operatoren-Vorrang und die Klammerung korrekt interpretiert wird.



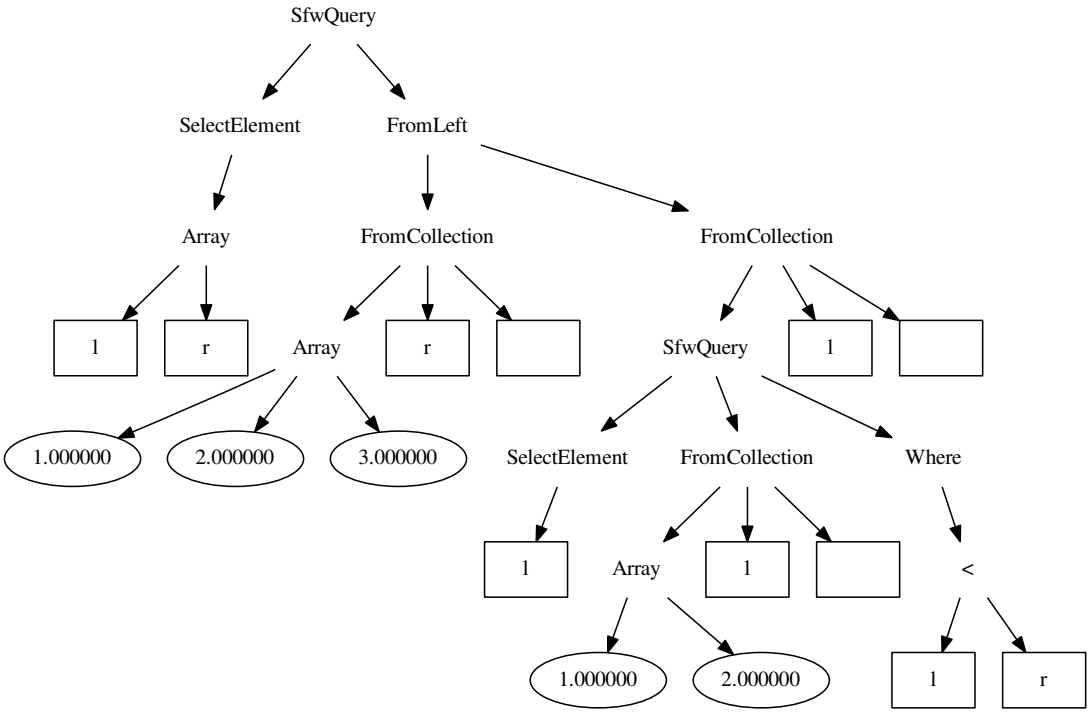
- **`SELECT "Hello" AS x, 123 AS y`**
  SFW-Query ohne FROM-Klausel. Die fehlende FROM-Klausel resultiert in einem `FromEmpty`-Objekt. Der syntaktische Zucker der SQL artigen SELECT-Liste wird erfolgreich in ein `SelectElement` von einem Tupel umgewandelt.



- **`SELECT ELEMENT a[2]`**
  `FROM [(SELECT ELEMENT nr FROM [0,1,2,3,4] AS nr)] AS a`
  Verschachtelter SFW-Query. Die Array-Navigation in der äußeren Query wurde korrekt gelesen. Die innere SFW-Query ist in einem Array-Konstruktor mit einem Element geschachtelt. Die leeren Boxen unter beiden `FromCollection`-Knoten resultieren aus den fehlenden `AT <var>` in den FROM-Klauseln.



- **`SELECT ELEMENT [l,r]`**
  `FROM [1,2] AS l RIGHT JOIN [1,2,3] AS r ON l < r`
  SFW-Query mit RIGHT JOIN. Der syntaktische Zucker der RIGHT-JOIN-Anweisung wurde korrekt in ein `FromLeft`-Objekt umgewandelt. Die Seiten wurden getauscht und eine Subquery generiert.



Kapitel 7

Ausführung

Die Ausführung findet in einem System statt welches eine Query im AST Format akzeptiert und einen Wert als Ergebnis liefert. In Abbildung 7.1 ist eine Übersicht der Ausführung dargestellt.

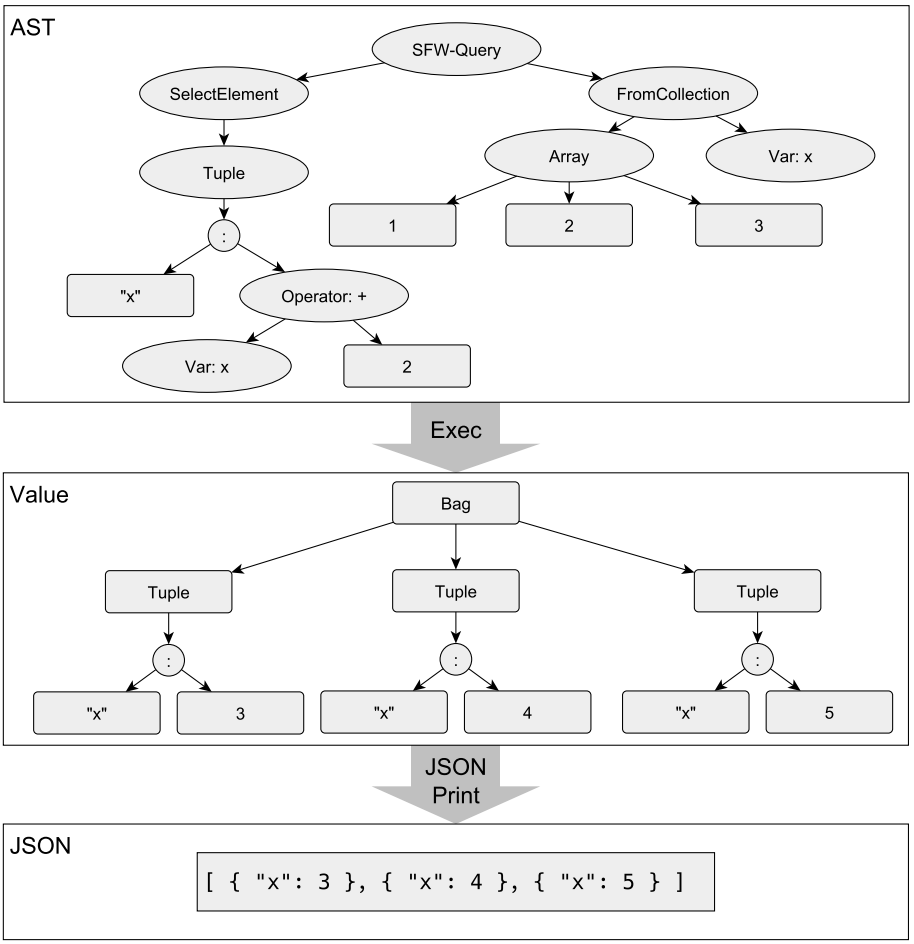


Abbildung 7.1: Query-Execution

Es müssen Funktionen geschrieben werden welche eine Expression evaluieren und den resultierenden Wert zurück geben. Da eine Expression eine Vereinigung einer Vielzahl unterliegender Typen ist müssen Funktionen zur Evaluation jeder dieser Typen geschrieben werden. Viele der Typen halten weitere Expressions, welche mit der gleichen Funktion evaluiert werden. Die Evaluierungsfunktionen arbeiten somit rekursiv und führen ein Tree-Traversal durch den AST durch. Dies bedeutet dass die Knoten im Baum besucht und evaluiert werden, wobei einige Teilbäume nie oder mehrfach besucht werden können.

7.1 Umgebungen

In der Spezifikation von SQL++ (Paragraph 2) wurde erläutert wie Ausdrücke in Umgebungen evaluiert werden. Dies spiegelt sich in den Evaluierungsfunktionen wieder. Jede der Funktionen akzeptiert eine Umgebung als Kontext der Ausführung. Eine Umgebung besteht aus Variablenbindungen, die entsprechen einem SQL++-Tupel in welchem Variablennamen an einen Wert gebunden werden. Die Umgebung legt somit fest zu welchem Wert eine Variable evaluiert wird.

Umgebungen werden außerdem häufig verkettet. Dabei wird aus zwei Umgebungen eine neue Umgebung gebildet welche die Variablenbindungen beider enthält. Wenn beide der verketteten Umgebungen eine gleichnamige Variable enthalten wird nur die Bindung aus einer übernommen, in der Verkettung $\Gamma_1|\Gamma_2$ dominieren die Variablenbindungen aus Γ_1.

Wenn eine Umgebung als einfaches `Tuple` implementiert ist müssen bei der Verkettung alle Variablenbindungen in dieses oder ein neues Objekt kopiert werden. Da verkettete Umgebungen oft nur temporär sind, wie wenn die Expression einer WHERE-Klausel in einer SFW-Query wiederholt evaluiert wird, ist das Manipulieren einer ursprünglichen Umgebung sowie das Generieren einer neuen Umgebung durch Kopieren der Variablenbindungen ineffizient.

Darstellung 7.1 Implementierung von Umgebung

```
struct Env {
  model::Tuple self;
  Env* next;
};
Env operator+(model::Tuple l, Env& r) { return Env(l, &r); }
```

Darstellung 7.1 zeigt die Implementierung einer Umgebung. Sie enthält ein Tupel sowie einen Pointer auf eine weitere Umgebung. Das Tupel hält die Variablenbindungen.

Wenn eine Umgebung durch Verkettung erstellt wird hält diese die Bindungen der linken Seite im Tupel und verweist auf die Umgebung der rechten Seite mit dem `next`-Pointer. Wenn in der Umgebung eine Variable gesucht wird wird das `self`-Tupel nach dieser durchsucht, sollte sie dort nicht gefunden werden wird der `next`-Pointer verfolgt und die Variable in der nächsten Umgebung gesucht. Dies erlaubt das Verketten von Umgebungen ohne Bindungen zu kopieren. Da nur die zuerst gefundene Bindung genutzt wird überlagert diese weitere Bindungen mit identischen Namen.

Der Operator `+` wurde definiert sodass aus einem Tupel `t` und einer Umgebung `e` die Verkettung durch `t + e` generiert werden kann.

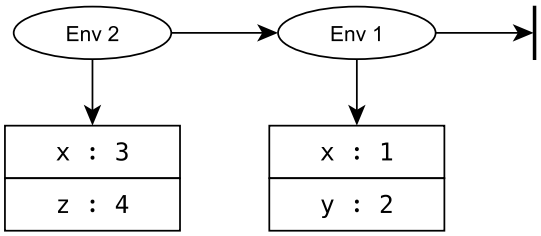


Abbildung 7.2: Struktur einer Umgebung

Die Struktur einer möglichen Umgebung ist in Abbildung 7.2 visualisiert. Die Umgebung `Env 2` ist dabei aus der Verkettung des Tupels `{ x:3, z:4 }` mit `Env 1` entstanden. Wenn die Variable `y` in `Env2` gesucht wird, wird die Suche in `Env 1` fortgesetzt und der Wert dort gefunden. Wenn jedoch die Variable `x` gesucht wird, wird der Wert `3` aus `Env 2` gefunden, womit die Belegung in `Env 1` überlagert wurde.

7.2 Expression

Eine Expression ist im AST als `variant` implementiert. Um auf das enthaltende Objekt zuzugreifen wird das Visitor-Pattern verwendet. Dabei wird ein Funktionsobjekt, der Visitor, übergeben und mit dem im `variant` enthaltenen Objekt aufgerufen. Darstellung 7.2 zeigt die Implementierung der Evaluierung einer Expression.

Wenn die Expression ein `Value` enthält wird vom Visitor die Funktion mit dem Typ `evalExpr(Value, Env, DbSession)` aufgerufen. Somit muss für jeden in einer Expression möglichen Typ eine `evalExpr`-Funktion definiert werden. Jede der Evaluierungsfunktionen akzeptiert den zu verarbeitenden Typ und gibt ein `Value`-Objekt als Ergebnis zurück.

Die `expr` und `env` Argumente werden als Referenzen übergeben (in C++ durch `&` dargestellt), womit die AST-Knoten und Umgebungen beim aufrufen der Evaluierungsfunk-

Darstellung 7.2 Evaluierung einer Expression

```
model::Value evalExpr(Expr& expr, Env& env, DbSession* session){
  return expr->apply_visitor(
    [&env, session](auto& obj) { return evalExpr(obj, env, session); });
}
```

tionen nicht kopiert werden. Das `session` Argument ist ein Pointer zu einem Objekt welches die Datenbankverbindung repräsentiert und wird später genauer erläutert. Die Argumente für die Umgebung und DB-Session bilden den Kontext der Evaluation und werden beim Aufruf von Evaluationsfunktionen von Unterelementen stets weiter gegeben.

7.3 Werte

Die Evaluierung einer Expression die ein SQL++-Value hält besteht ausschließlich aus dem Zurückgeben dieses Wertes (Darstellung 7.3).

Darstellung 7.3 Evaluierung eines Wertes

```
model::Value evalExpr(model::Value& val, Env&, DbSession*) {
  return val;
}
```

7.4 Variable

Bei der Evaluierung einer Variable muss die übergebene Umgebung nach dieser durchsucht werden. Da die Umgebung als Verkettung implementiert ist wird das Tupel der aktuellen Umgebung durchsucht. Sollte die Variable dort gefunden werden wird der entsprechende Wert zurückgegebene. Sonst wird die Suche in der nächsten Umgebung fortgesetzt.

Wenn die letzte Umgebung erreicht wurde und die Variable nicht erfolgreich gefunden wurde bestehen zwei Möglichkeiten. Es gibt einen Fehler in der Query und es wurde eine nicht definierte Variable genutzt oder es handelt sich um einen `named_value`. Da in der Spezifikation von SQL++ Variablen nicht klar von `named_value`s unterschieden werden können wird von einem `named_value` ausgegangen. Somit muss die Datenbank mit dem entsprechenden Namen angefragt werden, was im Paragraph 8 erklärt wird.

7.5 Komplexe Werte

Die komplexen Werte halten eine Menge von Expressions welche evaluiert werden müssen. In Darstellung 7.4 ist die Implementierung Array-Evaluierung dargestellt. Die Funktionen für `Tuple`, `Array` und `Bag` arbeiten ähnlich. Es wird ein Objekt vom zu konstruierenden Value-Typ angelegt. Anschließend werden alle enthaltenden Expressions durchlaufen und evaluiert. Die Umgebung und DB-Session wird weiter gegeben womit die Expressions im gleichen Kontext wie der komplexe Wert evaluiert werden. Die resultierenden `Value`s werden in das zuvor angelegte Objekt eingefügt, welches somit den komplexen Wert bildet welcher zurückgegeben wird.

Darstellung 7.4 Evaluierung von komplexen Werten

```
model::Value evalExpr(Array& array, Env& env, DbSession* session) {
  model::Collection output;
  output.has_order = true;
  for (auto& e : array)
    output.push_back(evalExpr(e, env, session));
  return output;
}
```

7.6 Navigationen

Die Array- und Tupel-Navigationen beginnen mit der Evaluierung der Basis-Expression. Der daraus resultierende Wert muss auf den korrekten Typ geprüft werden. Die Basis-Expression einer Array-Navigation muss eine `Collection` ergeben und die die Tupel-Navigation ein `Tuple`. Die Konfiguration `array_nav.allow_bag` entscheidet ob ein `Bag` als Basis einer Array-Navigation zugelassen ist. Sollte der Typ nicht übereinstimmen wird anhand von global gesetzten Konfigurationen eine Aktion ausgeführt. Anhand der Konfigurationen `array_nav.type_mismatch` und `tuple_nav.type_mismatch` wird entschieden ob ein Fehler ausgelöst wird oder `Null` oder `Missing` zurückgegeben wird.

In der Tupel-Navigation ist der Navigationsparameter ein Name und muss nicht weiter verarbeitet werden. Der aus der Basis-Expression resultierende Tupel wird nach diesem Namen durchsucht und der hinterlegte Wert zurückgegeben. Sollte der Name nicht im Tupel enthalten sein wird anhand der Konfiguration `tuple_nav.absent` zwischen Fehler, `Null` oder `Missing` entschieden.

In der Array-Navigation muss die Index-Expression evaluiert werden und der resultierende Wert auf den Typ `Number` geprüft werden. Ein falscher Typ resultiert im Verhalten welches in `array_nav.type_mismatch` beschrieben ist. Es wird geprüft ob die Zahl ein

valider Index im unterliegendem `vector` ist, also ob sie zwischen 0 und den enthaltenden Elementen liegt. Wenn der Index valide ist wird der Wert an dieser Position zurück gegeben oder im Fehlerfall das Verhalten aus `array_nav.absent` durchgeführt.

Alle Konfigurationen können im Quellcode angepasst werden. Die Standardwerte aller Konfigurationen sind Fehler gesetzt und Array-Navigation auf `Bag`s sind erlaubt.

7.7 Operatoren

Infix- und Prefix-Operatoren evaluieren die enthaltenen Expressions. Anhand des Operatores wird in einer Switch-Anweisung die den Operator implementierende Funktion aufgerufen und die aus den Expressions resultierenden Werte an diese Funktion übergeben.

Der Aufbau aller Operator-Funktionen ist ähnlich. In Darstellung 7.5 ist die Implementierung des Plus-Operators als Beispiel dargestellt. Es wird versucht den benötigten Typ aus den `Value`s zu extrahieren. Wenn dies erfolgreich ist wird die Operation durchgeführt und das Ergebnis zurückgegeben. Sollte die Umwandlung nicht möglich sein wird ein Fehler generiert.

Darstellung 7.5 Implementierung des Plus-Operators

```
model::Value opPlus(model::Value& l, model::Value& r) {
  model::Number* a = get<model::Number>(&l);
  model::Number* b = get<model::Number>(&r);

  if (a != nullptr && b != nullptr)
    return *a + *b;

  throw QueryError("Operator +: invalid operands");
}
```

7.8 SFW-Query

Die Funktionsweise von SFW-Queries ist in der Spezifikation genau beschrieben. Die FROM-Klausel generiert eine Menge von Binding-Tupeln welche durch die weiteren Klauseln manipuliert wird. Die Menge der Binding-Tupel ist in der Klasse `Bindings` als `vector<Tuple>` implementiert und entsprechend der `Collection`-Klasse mit einer booleschen Variable `has_order` versehen, die beschreibt, ob die Menge geordnet ist. Klauseln können die Ordnungs-Variable ändern und somit zwischen einer geordneten

und ungeordneten Menge umschalten. In der SELECT-Klausel wird die finale Menge von Binding-Tupeln in einen Wert umgewandelt welcher das Resultat der SFW-Query darstellt.

Darstellung 7.6 Implementierung der SFW-Query Evaluation

```
model::Value evalSfw(SfwQuery& sfw, Env& env, DbSession* session) {
  Bindings bindings = evalFrom(sfw.from, env, session);

  if (sfw.where)    applyWhere(  *sfw.where,    bindings, env, session);
  if (sfw.group_by) applyGroupBy(*sfw.group_by, bindings, env, session);
  if (sfw.having)   applyHaving( *sfw.having,   bindings, env, session);
  if (sfw.order_by) applyOrderBy(*sfw.order_by, bindings, env, session);
  if (sfw.limit || sfw.offset)
    applyLimitOffset(sfw.limit, sfw.offset, bindings, env, session);

  return applySelect(sfw.select, bindings, env, session);
}
```

Darstellung 7.6 zeigt die Implementierung der Evaluierung von SFW-Queries. `evalFrom` liefert die erste Menge von Binding-Tupeln wobei `applySelect` den Resultat-Wert liefert, welcher von der Funktion zurückgegeben wird. Alle dazwischen ausgewerteten Klauseln sind optional und manipulieren die Binding-Menge statt eine neue zu generieren. Die Evaluierungsfunktionen der Klauseln werden durch die `if`-Abfrage nur aufgerufen wenn diese Klausel existiert. Die `bindings`-Variable wird per Referenz übergeben, womit die ursprüngliche Binding-Menge in den Funktionen manipuliert werden kann. Das Anwenden der LIMIT- und OFFSET-Klauseln wird in einer Funktion durchgeführt da deren Verarbeitung miteinander verknüpft ist.

7.8.1 FROM

7.8.1.1 FromEmpty

Der speziell eingeführte `FromEmpty`-Knoten wird in Queries ohne FROM-Klausel verwendet. Dies ist im SQL++-Standard nicht vorgesehen und soll dem Verhalten von SQL entsprechen. Die Query `SELECT 1` in SQL resultiert in einem Record mit einer Spalte mit dem Wert `1`. Die entsprechende SQL++-Query ist `SELECT ELEMENT 1` und sollte das Ergebnis `{{ 1 }}` liefern (Bag mit einem Element). Da `SelectElement` für jedes Binding-Tupel die Expression evaluiert und ein Element im Ergebnis hinzufügt muss `FromEmpty` ein Binding-Tupel generieren. Das Binding-Tupel kann dabei leer sein. Darstellung 7.7 zeigt die Implementierung der Funktion die ein `Bindings`-Objekt aus einem leeren `Tuple` erzeugt und zurückgibt.

Darstellung 7.7 Implementierung der Evaluation von FromEmpty

```
Bindings evalFrom(FromEmpty&, Env&, DbSession*) {
  return { model::Tuple( ) };
}
```

7.8.1.2 FromCollection

Der `FromCollection`-Knoten resultiert aus der FROM-Klausel

```
FROM <expr> AS <var> AT <var>
```

Dabei muss die Expression zu einem Bag oder Array evaluiert werden. Die Funktion beginnt mit der Evaluierung der Expression und prüfen des Typs. Bei falschem Typ wird ein Fehler ausgelöst. Anschließend wird ein `Bindings`-Objekt angelegt. Die Bindings sind geordnet wenn die Expression ein Array ergeben hat.

Für jedes Element der Collection, welche aus der Expression resultiert, wird ein Binding-Tupel hinzugefügt. Das Element wird in das Binding-Tupel eingefügt, als Wert eines Paares mit dem Schlüssel aus `AS <var>`. Gleichzeitig wird eine Zähler inkrementiert der die Position des aktuell verarbeiteten Elements der Collection hält. Wenn der `AS <var>` Teil der Klausel gegeben ist, wird ein weiteres Paar in das Binding-Tupel eingefügt, welches aus dem Variablennamen und dem Zähler-Wert besteht.

Die Klausel

```
FROM ["a","b","c"] AS val AT nr
```

generiert somit die Binding-Tupel

- `{ val: "a", nr: 1 }`
- `{ val: "b", nr: 2 }`
- `{ val: "c", nr: 3 }`

7.8.1.3 FromTuple

Der `FromTuple`-Knoten resultiert aus der FROM-Klausel

```
FROM <expr> AS { <var> : <var> }
```

Die Expression muss dabei in ein Tupel evaluiert werden. Die Implementierung der Funktion ist in Darstellung 7.8 dargestellt.

Nach Evaluierung der Expression und prüfen des Types werden alle Paare des Tupels verarbeitet. Für jedes Paar wird ein Binding-Tupel generiert welches aus zwei Bindings

Darstellung 7.8 Pseudocode der Evaluierung von FromTuple

```
Bindings evalFrom(from_tuple, Γ, db_session)
    tuple ← evalExpr(from_tuple.expr, Γ, db_session)
    if tuple is no Tuple
        throw QueryError("FROM tuple: non-tuple")

    output ← Bindings()
    foreach (key, val) ← tuple
        binding ← Tuple()
        binding.insert(from_tuple.as_name, String(key))
        binding.insert(from_tuple.as_value, val)
        output.insert(binding)

    return output;
```

besteht. Der linke Variablenname referenziert den Schlüssel des Paares als String. Der zweite Variablenname referenziert den Wert des Paares.

Die Klausel

```
FROM { a: 1, b: 2} AS { name : val }
```

generiert somit die Binding-Tupel

- `{ name: "a", val: 1 }`
- `{ name: "b", val: 2 }`

7.8.1.4 FromInner

Der `FromInner`-Knoten resultiert unter anderem aus der FROM-Klausel

```
FROM <from> INNER CORRELATE <from>
```

Die Spezifikation beschreibt dass das rechte FROM-Item im Kontext jedes Binding-Tupels des linken evaluiert wird. Die aus dem rechten FROM-Item resultierenden Bindings bilden verbunden mit den Bindings welche als Kontext verwendet wurden ein Binding-Tuple der Ausgabe. Die Implementierung ist als Pseudocode in Darstellung 7.9 dargestellt.

In der Beispiel-Klausel

```
FROM [ { n:"a", nr:[1,2] },
       { n:"b", nr:[3] },
       { n:"c": nr:[] ,} ] AS x INNER CORRELATE x.nr AS y
```

Darstellung 7.9 Pseudocode der Evaluierung von FromInner

```
Bindings evalFrom(from_inner, Γ, db_session)
    left ← evalFrom(from_inner.left, Γ, db_session)
    output ← Bindings()
    foreach l ← left
        right ← evalFrom(from_inner.right, l | Γ, db_session)
        foreach r ← right
            binding ← Tuple()
            binding.insertAll(l)
            binding.insertAll(r)
            output.insert(binding)

    return output
```

wird das linke FROM-Item evaluiert und liefert die Binding-Tupel

- `{ x: { n:"a", nr:[1,2] } }`
- `{ x: { n:"b", nr:[3]   } }`
- `{ x: { n:"c", nr:[]    } }`

In einer Schleife wird jedes dieser Bindung-Tupel verarbeitet. Dabei wird aus der Verkettung des Binding-Tupels mit der aktuellen Umgebung eine neue Umgebung generiert. In dieser neuen Umgebung wird das rechte FROM-Item evaluiert.

Angenommen die aktuelle Umgebung besteht aus `{ z:42 }`. Durch Verkettung mit dem ersten Binding-Tupel wird somit die Umgebung `{ z:42, x: { n:"a", nr:[1,2] } }` generiert. In dieser wird das rechte FROM-Item evaluiert. Die Expression `x.nr` im rechten FROM-Item bezieht sich dabei aus die in der Umgebung definierte `x`-Variable mit dem Wert `{ n:"a", nr:[1,2] }`. Die Tupel-Navigation extrahiert aus dieser das `nr`-Element mit dem Wert `[1,2]` und das FROM-Item generiert die Binding-Tupel

- `{ y:1 }`
- `{ y:2 }`

Jedes dieser Binding-Tupel wird mit dem ersten Binding-Tupel des linken FROM-Items kombiniert und bildet ein Binding-Tupel der Ausgabe. In diesem Schritt werden die Binding-Tupel

- `{ x: { n:"a", nr:[1,2] }, y:1 }`
- `{ x: { n:"a", nr:[1,2] }, y:2 }`

generiert. Bei der Evaluierung des rechten FROM-Items mit dem zweiten Binding-Tupel wird das Binding-Tupel `{ x: { n:"b", nr:[3] }, y:3 }` generiert und zur Ausgabe hinzugefügt. Da in der Evaluierung mit dem letzten Binding-Tupel das Element

`nr` den Wert `[]` besitzt generiert das rechte FROM-Item eine leere Menge von Binding-Tupel womit keine Bindings zur Ausgabe hinzugefügt werden.

7.8.1.5 FromLeft

Der `FromLeft`-Knoten resultiert unter anderem aus der FROM-Klausel

```
FROM <from> LEFT OUTER CORRELATE <from>
```

Die Evaluierung verläuft nahezu identisch der Evaluierung von `FromInner`. Wenn das rechte FROM-Item eine leere Menge von Binding-Tupel zurück gibt wird dennoch ein Binding-Tupel in der Ausgabe generiert. Wie in der als Pseudocode dargestellten Implementierung in Darstellung 7.10 zu sehen ist wird wenn die Menge der Binding-Tupel aus dem rechten FROM-Item leer ist das linke Binding-Tupel direkt eingefügt.

Darstellung 7.10 Pseudocode der Evaluierung von FromLeft

```
Bindings evalFrom(from_left, Γ, db_session)
    left ← evalFrom(from_left.left, Γ, db_session)
    output ← Bindings()
    foreach l ← left
        right ← evalFrom(from_left.right, l | Γ, db_session)
        if not right.empty()
            foreach r ← right
                binding ← Tuple()
                binding.insertAll(l)
                binding.insertAll(r)
                output.insert(binding)
        else
            output.insert(l);

    return output
```

Das letzte Binding-Tupel des vorherigen Beispiels ist `{ x: { n:"c", nr:[] } }`. Das linke FROM-Item liefert bei der Evaluierung in diesem Kontext keine Ergebnisse wodurch in `FromInner` kein Element zur Ausgabe hinzugefügt wird. In der Evaluierung von `FromLeft` wird in diesem Fall das Tupel allein der Ausgabe hinzugefügt.

7.8.1.6 FromFull

Der `FromFull`-Knoten resultiert unter anderem aus der FROM-Klausel

```
FROM <from> FULL OUTER CORRELATE <from> ON <expr>
```

Beide FROM-Items sind separat zu evaluieren. Alle Kombinationen aus einem linken und rechten Binding-Tupel werden in die Ausgabe eingefügt wenn mit diesen die Expression zu Wahr ausgewertet wird. Binding-Tupel beider Seiten welche nicht in der Ausgabe vorkommen werden einzeln Hinzugefügt.

Die Klausel

```
FROM [1,2,3] AS l FULL OUTER CORRELATE [1,2,3] AS r ON l < r
```

sollte die Binding-Tupel

- `{ l:1, r:2 }`
- `{ l:1, r:3 }`
- `{ l:2, r:3 }`
- `{ l:3     }`
- `{     r:1 }`

generieren.

Wie in Darstellung 7.11 zu sehen kann die Evaluierung durch zwei verschachtelte Schleifen implementiert werden. Für alle Binding-Tupel der linken Seite werden alle Binding-Tupel der rechten Seite betrachtet. Die Expression wird in der Umgebung `l | r | Γ` evaluiert, eine Verkettung aus dem linken und rechtem Tupel sowie der ursprünglichen Umgebung. Wenn die Expression zu `true` evaluiert wird, wird die Kombination beider Tupel der Ausgabe hinzugefügt.

Zusätzlich müssen die nicht genutzten Tupel erkannt werden. Die äußere Schleife verarbeitet die Menge der linken Binding-Tupel. In jedem Schleifendurchlauf wird eine boolesche Variable erstellt welche festhält ob das aktuelle Tupel in der Ausgabe genutzt wurde. Diese ist auf `false` initialisiert und wird wenn in der inneren Schleife das Tupel zum Ergebnis hinzugefügt wird auf `true` gesetzt. Nach dem Ausführen der inneren Schleife wird die Variable geprüft, sollte das Tupel nicht verwendet wurden sein wird es einzeln der Ausgabemenge hinzugefügt.

Um die nicht genutzten Tupel der rechten Menge festzuhalten wird ein Feld mit booleschen Variablen erstellt, welches die genutzten Tupel markiert. Die Größe des Feldes entspricht der Anzahl rechter Binding-Tupel und jeder Wert ist auf `false` initialisiert. Wenn ein Tupel zu der Ausgabe hinzugefügt wird, wird die Variable im Feld an dieser Position auf `true` gesetzt. Nach Abschluss der äußeren Schleife wird das Feld iteriert und alle nicht verwendeten Tupel der rechten Menge einzeln der Ausgabe hinzugefügt.

Darstellung 7.11 Pseudocode der Evaluierung von FromFull

```
Bindings evalFrom(from_full, Γ, db_session) {
    left ← evalFrom(from_full.left, Γ, db_session);
    right ← evalFrom(from_right.right, Γ, db_session);
    right_used ← Array<Bool>()
    right_used.fill(false, right.size())
    foreach l ← left
        left_used ← false;
        for i ← 0 to right.size() - 1
            r ← right[i];
            cond ← evalExpr(from_full.cond, l | r | Γ, db_session)
            if cond is no Bool
                throw QueryError("FROM FULL: non-bool condition")
            if cond == true
                binding ← Tuple()
                binding.insertAll(l)
                binding.insertAll(r)
                output.insert(binding)
                right_used[i] ← true
                left_used ← true
        if not left_used
            output.insert(l)
    for i ← 0 to right.size() - 1
        if not right_used[i]
            output.insert(right[i])
    return output;
```

7.8.2 WHERE

Die WHERE-Klausel ist die erste der Klauseln welche die Binding-Tupel manipulieren. In der Evaluierung werden die Tupel als Referenz übergeben und entsprechend geändert. Es sollen alle Tupel entfernt werden in deren Kontext die enthaltende Expression nicht zu `true` evaluiert wird.

In der Implementierung wird dazu die `remove_if` Funktion der Standardbibliothek genutzt. Diese akzeptiert eine Menge und eine Funktion als Argument. Die Funktion wird mit jedem Element der Menge aufgerufen. Wenn die Funktion `true` zurückgibt wird das entsprechende Element entfernt. Die verwendete Funktion evaluiert die in der FROM-Klausel enthaltende Expression im Kontext des übergebenen Tupels verkettet mit der ursprünglichen Umgebung. Wenn der dabei erhaltende Wert nicht vom Typ `Bool` ist wird ein Fehler ausgelöst, anderenfalls wird die Negation des Wertes zurückgegeben. Ein Tupel mit welchem die Expression `false` liefert bewirkt dass die Hilfsfunktion `true` ergibt, womit dieses Tupel durch `remove_if` aus der Menge entfernt wird.

7.8.3 GROUP BY

Die GROUP-BY-Klausel enthält eine Menge von Termen, jeder Term besteht aus einer Expression sowie einem optionalen Variablennamen (`GROUP BY genre AS g, subgenre AS sg`). Die Binding-Tupel sollen in Gruppen eingeteilt werden in welchen alle Binding-Tupel die gleichen Werte bei der Evaluation der Expressions generieren. Nach der Verarbeitung von GROUP BY soll für jede Gruppe ein Binding-Tupel existieren in welchem der Name `"group"` enthalten ist. `"group"` verweist auf einen Bag aller in dieser Gruppe enthaltenden Binding-Tupel.

Die Binding-Tupel

- `{ name: "Mozart", genre: "classic" }`
- `{ name: "Pink Floyd", genre: "rock" }`
- `{ name: "Beethoven", genre: "classic" }`

ergeben, nach der Verarbeitung der Klausel `GROUP BY genre AS g`, die Binding-Tupel

- ```
  { g: "classic", group: {{
    { name: "Mozart", genre: "classic" },
    { name: "Beethoven", genre: "classic" }
  }} }
  ```
- ```
  { g: "rock", group: {{
    { name: "Pink Floyd", genre: "rock" }
  }} }
  ```

Darstellung 7.12 zeigt den Pseudocode der Evaluierungsfunktion. Da die Gruppenzugehörigkeit von mehreren Expressions, und somit mehreren Werten, abhängt wird der `GroupKey` Typ eingeführt. Dieser enthält eine Menge von Werten und stellt Vergleichsoperatoren bereit. Beim Vergleich von zwei `GroupKey`s werden die Werte in der enthaltenen Reihenfolge verglichen, die ersten sich unterscheidenden Werte dienen als Ergebnis des Vergleichs.

Es wird eine `Map` angelegt welche einen `GroupKey` auf einen Bag (`Collection`) abbildet. Aus jedem Binding-Tupel wird, durch Evaluierung der Expressions in der entsprechenden Umgebung, ein `GroupKey` generiert. Dieser `GroupKey` wird in der `Map` gesucht und das Binding-Tupel in den gefundenen Bag eingefügt.

Nach der Gruppierung der Binding-Tupel wird die zu manipulierende Menge von Binding-Tupeln geleert. Für jede Gruppe wird ein neues Binding-Tupel in die Menge eingefügt, welches den Namen `"group"` und den zugehörigen Bag enthält. Zusätzlich

wird für jeden Term der GROUP-BY-Klausel, welcher einen Namen enthält, der Wert der Gruppierungsexpression an diesen Namen gebunden.

Darstellung 7.12 Pseudocode der Evaluierung von GroupBy

```
void applyGroupBy(group_by, bindings, Γ, db_session) {
    groups ← Map<GroupKey, Collection>()
    foreach b ← bindings
        key ← GroupKey();
        foreach term ← group_by
            key.insert(evalExpr(term.expr, b | Γ, db_session))
        groups[key].insert(b)

    bindings.clear()
    bindings.has_order = false

    foreach (key, vals) ← groups
        binding = Tuple()
        binding.insert("group", vals)
        for i ← 0 to group_by.size() - 1
            if group_by[i].as not empty
              binding.insert(group_by[i].as, key[i])
        bindings.insert(binding)
```

7.8.4 HAVING

Die `HAVING`-Klausel ist identisch zu der `WHERE`-Klausel implementiert. Die Klauseln unterscheiden sich nur im Zeitpunkt der Evaluierung. `HAVING` wird nach `GROUP BY` evaluiert, womit sich die enthaltene Expression auf das spezielle `"group"`-Binding beziehen kann. Aggregatfunktionen, wie Summe oder Durchschnitt, können in der `HAVING`-Klausel genutzt werden.

7.8.5 ORDER BY

Die `ORDER-BY`-Klausel besteht aus einer Menge von Termen. Jeder Term besteht aus einer Expression sowie einer booleschen Variable die Beschreibt ob die Ordnung ab- oder aufsteigend ist. Die Binding-Tupel sollen anhand dieser Terme sortiert werden.

In der Implementierung wird die `sort` Funktion der Standardbibliothek genutzt. Diese wird mit einer selbst definierten Vergleichsfunktion aufgerufen, welcher zwei Elemente übergeben werden und `true` zurück gibt wenn das erste Argument kleiner ist. Die Vergleichsfunktion verarbeitet die Terme der Klausel, für jeden Term wird die Expression in den Umgebungen der zu vergleichenden Binding-Tupel evaluiert. Sollten die Werte

gleich sein wird der nächste Term verarbeitet, sonst wird das Resultat des implementierten `<`-Operators zurückgegeben, welches invertiert wird wenn der Term als `DESC` (absteigend) markiert ist. Die `has_order` Variable der Menge der Binding-Tupel wird auf `true` gesetzt, da durch `ORDER BY` eine Ordnung festgelegt wurde.

Es ist jedoch möglich dass eine Expression, welche in unterschiedlichen Umgebungen ausgewertet wird, Ergebnisse von verschiedenen Typen liefert. `ORDER BY` sollte auch für heterogene Datenmengen ein angemessenes Resultat liefern. Die Spezifikation von SQL++ trifft dazu keine Aussage. Die Vergleichsoperatoren werden angepasst um ein Ergebnis beim Aufruf mit verschiedenen Typen zu liefern, womit eine Ordnung der Typen festgelegt wird. Diese Ordnung wurde auf die Reihenfolge der Definition in `Value` festgelegt und ist: `Missing < Null < Number < Bool < String < Tuple < Array < Bag`.

7.8.6 LIMIT und OFFSET

Die `LIMIT`- und `OFFSET`-Klauseln halten je eine Expression. Diese werden, wenn gegeben, evaluiert und der Rückgabewert auf den Typ `Number` geprüft. Der Wert der `OFFSET`-Expression beschreibt die Anzahl der Binding-Tupel welche vom Beginn der Menge entfernt werden. Der Wert der `LIMIT`-Expression beschreibt die Anzahl der Binding-Tupel welche anschließend in der Menge gehalten werden, alle weiteren werden entfernt.

7.8.7 SELECT

7.8.7.1 SELECT ELEMENT

Die `SELECT-ELEMENT`-Klausel besteht aus einer Expression. Es wird eine `Collection` als Rückgabewert angelegt und die `has_order`-Variable aus der Menge der Binding-Tupel übernommen. Die Expression wird im Kontext jedes Binding-Tupels evaluiert und das Ergebnis, wenn es nicht vom Typ `Missing` ist, zu der Ausgabe hinzugefügt.

Mit den Binding-Tupeln

- `{ a: 1, b: 2 }`
- `{ a: 3, b: 4 }`

ergibt `SELECT ELEMENT a + b` als Ausgabe den Bag

```
{{ 3, 7 }}
```

7.8.7.2 SELECT ATTRIBUTE

Die `SELECT-ATTRIBUTE`-Klausel besteht aus zwei Expressions. Als Rückgabewert wird ein Tupel angelegt. Beide Expressions werden im Kontext jedes Binding-Tupels evaluiert und ein Paar aus den Werten beider Expressions zum Ausgabetupel hinzugefügt. Der Wert der Expression welche den Namen bildet muss dabei ein String sein, sonst wird ein Fehler ausgelöst.

Mit den Binding-Tupeln

- `{ name: "Albert", age: 35 }`
- `{ name: "Bertha", age: 50 }`

ergibt `SELECT ATTRIBUTE name : age` als Ausgabe das Tupel

```
{ Albert: 35, Bertha: 50 }
```

7.9 Funktionen

Funktionsaufrufe bestehen aus einem Funktionsnamen und einer Menge von Expressions, welche die Argumente der Funktion bilden. In der Evaluation wird der Funktionsname zu einer aufrufbaren Funktion aufgelöst. Die Expressions werden evaluiert und aus den Ergebnissen wird eine Menge von Werten gebildet. Mit dieser Menge von Werten wird die Funktion aufgerufen, welche die Anzahl und den Typ der Werte überprüft, die entsprechende Funktionalität ausführt und das Ergebnis liefert.

Diese Implementierung ist jedoch nicht vollständig, da Aggregatfunktionen existieren welche in einer SQL-artigen Weise nutzbar sein sollen. Die Spezifikation beschreibt dass Aggregatfunktionen welche in `HAVING`, `ORDER BY` oder `SELECT` nach einem `GROUP BY` verwendet werden spezielle Regeln besitzen. So soll der Aufruf `f(` e `)` im Gruppen-Kontext `f(SELECT` e' `FROM group AS g)` entsprechen, wobei e' der Expression e entspricht in der jeder Name n_i durch $g.n_i$ ersetzt ist. Somit soll `SUM(x)` nach einer Gruppierung `SUM(SELECT g.x FROM group AS g)` entsprechen.

Bei der Evaluierung von `SUM(p.age)`, z.B. als Teil der Query `SELECT ELEMENT SUM(p.age) FROM people AS p GROUP BY p.gender`, im Kontext des Binding-Tupels

```
{ group: {{
    { p: { name: "Albert", gender: "m", age: 35 } },
    { p: { name: "Edward", gender: "m", age: 40 } }
  }} }
```

wird an die Funktion `SUM` das Argument

```
{{ 35, 40 }}
```

übergeben. Da dieser Wert auch bei dem Aufruf `SUM(SELECT g.p.age FROM group AS g)` übergeben wird.

Da beim Parsing nicht bekannt ist welche Funktionen existieren und festzustellen ob die Funktion im Kontext einer Gruppierung steht kompliziert ist wird die spezielle Behandlung von Aggregatfunktionen in der Evaluierung implementiert.

Eine Evaluation steht im Kontext eines `GROUP BY` wenn das Binding-Tupel in welchem evaluiert wird ein `"group"`-Element enthält. In der Evaluierung einer Expression hat dies zur Folge, dass in der neusten Umgebung der `"group"`-Name an eine Collection von Tupeln gebunden ist. Statt eine Subquery zu generieren kann die entsprechende Funktionalität direkt implementiert werden. Dabei wird mit jedem Tupel der Gruppe die als Argument gegebene Expression evaluiert und die Ergebnisse in einem Bag gesammelt. Dieser Bag wird als Argument an die eigentliche Funktion übergeben.

In Darstellung 7.13 ist die Implementierung als Pseudocode gegeben. Zu beginn wird versucht den Funktionsnamen zu einer normalen Funktion aufzulösen, wenn dies gelingt werden die Expressions evaluiert, die Argumente generiert und die Funktion aufgerufen. Sollte es sich nicht um eine gewöhnliche Funktion handeln wird versucht den Namen zu einer Aggregatfunktion aufzulösen. Wenn dies gelingt wird der `"group"`-Name in der zuletzt generierten Umgebung gesucht und der Typ dieser geprüft. Die Tupel der `"group"` werden als Kontext für die Argument-Expression genutzt und der Bag der Werte aufgebaut mit welchem die Aggregatfunktion aufgerufen wird. Wenn `"group"` nicht gefunden wurde, wird die Expression gewöhnlich evaluiert und die Funktion mit diesem Wert aufgerufen. Somit wird ermöglicht dass Aggregatfunktionen sowohl nach `GROUP BY` sowie auf direkt gegebene Mengen (`SUM([1,2,3])`) angewandt werden können.

Darstellung 7.13 Pseudocode der Evaluierung von Funktionen

```
Value evalFunction(func, Γ, db_session)
    if callable ← functions.find(func.name) // regular function
        args ← Array<Value>()
        foreach arg_expr ← func.arguments
            args.insert(evalExpr(arg_expr, Γ, db_session))
        return callable(args)

    if callable ← aggregte_functions.find(func.name) // aggregate function
        if func.arguments.size() != 1
            throw QueryError("Aggregate functions require 1 argument")

        if group ← Γ.first.find("group") // most recent env has "group"
            coll ← Collection()
            foreach tuple ← group
                coll.insert(evalExpr(func.arguments[0], tuple | Γ,
                                     db_session))
            return callable(coll)

        else // most recent env has no "group"
            coll = evalExpr(func.arguments[0], Γ, db_session)
            if coll is no Collection
                throw QueryError("Aggregate functions require collection"
                                 " or GROUP BY")
            return callable(coll)

    throw QueryError("Unknown function: " + func.name);
```

Kapitel 8

Datenanbindung

Die Queryausführung soll im Kontext der Datenbank stattfinden. In SQL++ können Queries durch `named_value`s auf die Datenbank zugreifen.

8.1 Deadlocks

Da die implementierte Query-Language ausschließlich lesend auf die Datenbank zugreift und Schreiboperationen in *Cheesebase* nur einzelne Werte betreffen können keine Deadlocks auftreten. Alle Objekte in der Datenbank sind mit zweistufigen Read-Write-Locks versehen. Während einer Queryausführung werden nur Read-Locks angefordert und gehalten, womit mehrere Queries problemlos parallel ausgeführt werden können. Gleichzeitig erlaubt *Cheesebase* dass in einer atomaren Schreiboperation nur ein Objekt geschrieben werden kann.

Sollten zukünftig komplexere Schreiboperationen implementiert werden muss ein System für das Vermeiden oder Auflösen von Deadlocks hinzugefügt werden.

8.2 Einfache Implementierung

In der Evaluierung einer Query wird ein Pointer zu einem `DbSession`-Objekt akzeptiert und an alle aufgerufenen Evaluierungsfunktionen weiter gegeben. In der Evaluierung von Variablen muss, wenn die Variable nicht in der Umgebungen gefunden wird, der Wert mit diesem Namen aus der Datenbank gelesen werden.

Ein `DbSession`-Objekt besteht aus einem `disk::ObjectR`-Objekt, dies ist eine interne read-only Repräsentation eines Tupels welche eine Referenz zur Datenbank, die interne

Speicheradresse des Tupels sowie das Read-Lock dieses Tupels enthält. Dieses Objekt kann genutzt werden um Unterelemente anzufordern oder den gesamten Wert zu lesen. In einer `DbSession` ist das `disk::ObjectR`-Objekt der Wurzel der geöffneten Datenbank enthalten. Ausserdem wird eine `getNamedValue(string name)` Methode implementiert, welche den entsprechenden Namen aus dem Wurzel-Tupel der Datenbank anfordert.

Somit kann beim evaluieren einer Variable das entsprechende `named_value`, welches ein Element des Wurzel-Tupels der Datenbank ist, gelesen und zurückgegeben werden. Dieses Vorgehen ist sehr naiv und führt dazu dass das gesamte `named_value` gelesen wird, auch wenn nur ein Unterelement dieses benötigt wird.

In der Schnittstelle von *Cheesebase* wurde eine `query`-Funktion hinzugefügt, welche in Darstellung 8.1 dargestellt ist.

Darstellung 8.1 Query-Funktion in Cheesebase

```
Value Cheesebase::query(string& query_string) {
  auto query = parseQuery(query_string);
  auto session = DbSession(database);
  return evalQuery(query, &session);
}
```

8.3 Beispiel

Zum Testen der Datenanbindung wird eine Datenbank angelegt und Testdaten in diese Eingefügt. Die Testdaten sind ein Array namens `"people"` mit 6 Einträgen der Form:

```
people: [ {
    "fname": "Dave",
    "lname": "Smith",
    "gender": "m",
    "age": 46,
    "children": [ { "age": 17, "fname": "Aiden", "gender": "m" },
                  { "age": 12, "fname": "Bill",  "gender": "f" } ],
    "email": "dave@gmail.com",
    "hobbies": [ "golf", "surfing" ],
  }, ... ]
```

In dieser Datenbank wurde die Query

```
SELECT p.fname AS first, p.lname AS last FROM people AS p
```

ausgeführt.

Die Query liefert das Resultat:

```
[ { "first": "Dave",  "last": "Smith" },
  { "first": "Earl",  "last": "Johnson" },
  { "first": "Fred",  "last": "Jackson" },
  { "first": "Harry", "last": "Jackson" },
  { "first": "Ian",   "last": "Taylor" },
  { "first": "Jane",  "last": "Edwards" } ]
```

Um die Effizienz der Query-Ausführung einzuschätzen wurde ein Zähler implementiert der alle in der Datenbank gelesenen Werte zählt. Komplexe Werte, wie ein Array, werden dabei einmal selbst gezählt und jedes gelesene Unterelement ein weiteres Mal.

Das zuvor dargestellte Beispiel liest mit der einfachen Implementierung 93 Werte aus der Datenbank. Diese hohe Zahl resultiert daraus, dass der Zugriff auf den `named_value people` das gesamte Array, inklusive aller Unterelemente wie `"children"` und `"hobbies"`, liest. Aus diesem Grund wird versucht eine Implementierung zu finden welche nur benötigte Daten aus der Datenbank liest.

8.4 Lazy Implementierung

In der Evaluation wird mit `Value`s agiert. Eine Expression der Form `people[3]` wird evaluiert indem das Array welches an `"people"` gebunden ist generiert wird und darauf folgend das Unterelement mit dem Index `3` referenziert wird. Jedoch ist es möglich das ein Großteil des `"people"`-Array nicht benötigt wird.

Um dies zu vermeiden werden die komplexen `Value`s *lazy* implementiert. *Lazy-evaluation* beschreibt dass ein Wert erst generiert wird wenn dieser benötigt wird. In der Datenanbindung der Query-Evaluierung bedeutet dies dass `Tuple` und `Collection`, wenn sie von der Datenbank gelesen wurden, nur durch eine Referenz in die Datenbank repräsentiert werden. Erst wenn der Wert verwendet wird findet das Lesen in der Datenbank statt. Wenn nur ein Unterelement genutzt wird, kann dieses gezielt aus der Datenbank angefordert werden ohne den gesamten Wert zu lesen.

`Tuple` und `Collection` halten je ein Objekt vom Typ `variant<Tuple_base, Tuple_lazy>` oder `variant<Collection_base, Collection_lazy`. Der `base`-Typ entspricht der ursprünglichen Implementierung des Wertes. Der `lazy`-Typ enthält eine Datenbankreferenz (`disk::ObjectR` oder `disk::ArrayR`). Beide Varianten des Tupels sind minimal in Abbildung 8.1 dargestellt.

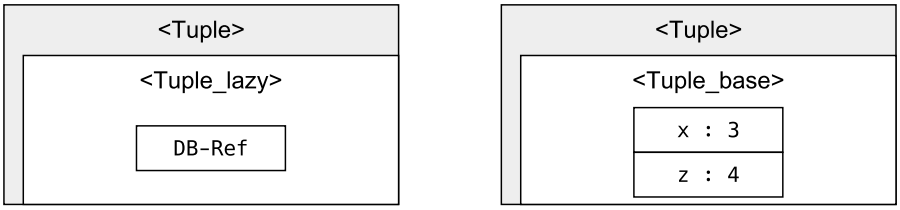


Abbildung 8.1: Varianten eines Tupel

`Tuple` und `Collection` stellen Methoden bereit um auf ein bestimmtes Unterelement zuzugreifen oder die gesamte Menge zu Iterieren. Diese Methoden leiten auf das im `variant` enthaltende Objekt weiter. Der Zugriff auf ein bestimmtes Unterelement bewirkt dass dieses aus dem `base`-Objekt zurückgegeben wird oder durch das `lazy`-Objekt eine Anfrage an die Datenbank gestellt wird. Beim Iterieren der gesamten Menge wird dies direkt an das `base`-Objekt weitergegeben, wenn es sich um ein `lazy`-Objekt handelt wird dieses komplett aus der Datenbank gelesen und das im `variant` gehaltene Objekt in ein `base`-Objekt umgewandelt. Somit wurde das Verhalten der Klassen bei direkt enthaltenen Werten nicht geändert. Wenn es sich um ein Element aus der Datenbank handelt, wird dieses entweder erst beim kompletten Zugriff gelesen oder bei Abfrage von bestimmten Unterelementen nur diese aus der Datenbank gelesen.

Die `lazy`-Klasse enthält ausserdem einen Cache für gelesene Werte. Wenn das gleiche Unterelement mehrmals abgefragt wird, findet das lesen der Datenbank nur bei der ersten Abfrage statt.

Das DBMS selbst wurde angepasst um beim Lesen, wenn möglich, `lazy`-Werte zu liefern. Um lange Datenbank-Sessions zu minimieren wurde eine `fetch`-Methode implementiert, welche rekursiv alle `lazy`-Werte auflöst und in gewöhnliche Werte umwandelt. Diese wird auf das finale Ergebnis einer Query angewandt, bevor dieses zum Benutzer zurückgegeben wird.

Mit dieser Implementierung wurde das zuvor vorgestellte Beispiel erneut durchgeführt. Das gleiche Ergebnis konnte mit 19 gelesenen Werten (statt 93) generiert werden. Im Beispiel wurde das `"people"`-Array, die 6 enthaltenen Tupel und je 2 Strings pro Tupel gelesen, woraus die 19 (1 + 6 + 6*2) gelesenen Werte resultieren.

Die Query

```
SELECT ELEMENT p.fname FROM people AS p WHERE p.age < 30
```

liefert zwei Namen und wurde mit 15 gelesenen Werten ausgeführt (`"people"`-Array + 6 Personen-Tupel + 6 `"age"`-Elemente + 2 Namen).

Kapitel 9

Mögliche Optimierungen

Im Folgendem werden Ansätze vorgestellt wie die Query-Ausführung weiter optimiert werden kann.

9.1 Statische Teilbäume

Die Query

```
SELECT ELEMENT p.name
          FROM people AS p
         WHERE p.age > AVG(SELECT p.age FROM people AS p)
```

gibt die Namen aller Personen aus deren Alter über dem Durchschnitt liegt. Die Expression der `WHERE`-Klausel wird für jedes in `people` enthaltene Element evaluiert. Durch das Caching der `named_value`s, welches mit der Lazy-evaluierung implementiert wurde, finden keine überflüssigen Datenbankanfragen statt. Jedoch wird die `AVG`-Funktion wiederholt aufgerufen und der Durchschnitt berechnet.

In einer optimierten Query-Ausführung sollten Expressions nur neu evaluiert werden wenn eine genutzte Variable einen anderen Wert besitzt. Es kann festgestellt werden von welchem Kontext eine Expression abhängt. Die Expression in der `WHERE`-Klausel (`p.age > AVG(...)`) hängt von der in der `FROM`-Klausel definierten Variable `p` ab. Die Unterexpression `AVG(...)` ist jedoch von keiner ausserhalb definierten Variable abhängig. Die Abhängigkeiten in einer Query müssen, eventuell vor der Ausführung, ermittelt werden. Mit diesen Informationen kann in der Ausführung entschieden werden ob Teilbäume neu evaluiert werden müssen oder das Ergebnis nach der ersten Evaluierung zwischengespeichert werden soll.

9.2 Indizes

Indizes können in Document-Stores implementiert werden. Dazu muss jedoch ein Schema eingeführt werden, welches die Werte an ausgewählten Positionen einschränkt. Wenn ein Index für Daten implementiert wird deren Struktur an einer SQL-Tabelle orientiert ist, muss eingeschränkt werden dass es sich um ein Array handelt welches ausschließlich Tupel enthält. Der indexierte Wert sollte ausserdem in jedem Tupel einen festgelegten Typ besitzen, da ein Index meist über einem einzigen Datentyp implementiert wird. Indizes über heterogene Daten sind prinzipiell möglich, benötigt jedoch angepasste Datenstrukturen.

In der SQL++-Query-Ausführung können Indizes bei der Evaluierung von SFW-Queries genutzt werden. Dabei muss nach bestimmten Mustern der Klauseln gesucht werden, die das Ausnutzen von Indizes erlauben. In der Query `SELECT ELEMENT p.name FROM people AS p WHERE p.age < 30` muss erkannt werden dass die `WHERE`-Klausel dem Muster `<member> <operator> <static-expression>` folgt. Aus diesem Muster kann erkannt werden dass in der `FROM`-Klausel eine Range-Anfrage auf `people` ausreichend ist, welche einen Index nutzen kann. Die Mustererkennung in Klauseln wird auch von relationalen Query-Engines verwendet, wie in der WHERE-Klausel-Analyse von *SQLite* [9].

Kapitel 10

Fazit

10.1 Zusammenfassung

In diesere Arbeit wurde eine NoSQL-Query-Language implementiert und in das selbst entwickelte DBMS *Cheesebase* eingebunden.

Als Query-Language wurde SQL++ gewählt, eine sehr junge Spezifikation die versucht die Semantik anderer Sprachen zu vereinen. Es wurde gezeigt wie ein Interpreter für SQL++ implementiert und in das DBMS eingebunden wird. Dabei wurde die *Cheesebase*-Bibliothek um eine Query-Schnittstelle erweitert die es ermöglicht SQL++-Queries auf einer Datenbank mit JSON-orientierten Daten auszuführen. Die in SQL++ definierten Features wurden fast vollständig implementiert, es fehlen Set-Operationen (`UNION`, `INTERSECT` und `EXCEPT`).

10.2 Beispiele

SQL++-Queries welche korrekt in *Cheesebase* ausgeführt werden können enthalten:

- Berechnungen mit Vorzeichenvorrang

  ```
  1 / 2 + 3 * 4
  ```

- Abfrage von speziellen Werten

  ```
  people[4].name
  ```

- Abfragen aus Tabellen-artigen Werten

```
SELECT  p.name AS name,
        p.age AS age,
        p.gender AS gender
  FROM  people AS p
```

- Gruppierung und Aggregatfunktionen

```
SELECT  g AS gender,
        AVG(p.age) AS average_age
  FROM  people AS p
GROUP BY  p.gender AS g
```

- Filter, Subqueries und Sortieren

```
SELECT  p.name AS name
  FROM  people AS p
 WHERE  p.age < AVG(SELECT ELEMENT p.age FROM people AS p)
ORDER BY  p.age
```

- Joins

```
SELECT  o.product AS product,
        o.amount  AS amount,
        s.name    AS supplier
  FROM  orders AS o
INNER JOIN  suppliers AS s
        ON  s.id == o.supplier
```

10.3 Einschätzung von SQL++

Auch wenn SQL++ nicht alle Features von SQL unterstützt, wie das Definieren von Namen in SELECT-Klauseln welche im gleichen Query verwendet werden können, ist die Ähnlichkeit zu SQL einer der größten Vorteile gegenüber anderer semi-strukturierten Query-Languages. SQL++ kann von mit SQL vertrauten Personen intuitiv und schnell gelernt werden, was angesichts der großen Anzahl von SQL Entwicklern einen hohen Wert besitzt. Die hohe Kombinierbarkeit innerhalb einer Query ist ein weiterer positiver Aspekt. Das Nutzen von beliebigen Expressions in allen Teilen einer Query, inklusive als Quelle der FROM-Klausel, erlaubt eine große Freiheit beim Verfassen von Queries.

In der Definition der Sprache sind Datenflüsse der Query-Ausführung klar dargestellt, was bei der Implementierung zur Hilfe kommt. Es gibt wenig Probleme beim Realisieren der Semantik, womit sich SQL++ anbietet von anderen Document-Store-Systemen implementiert zu werden.

Literatur

[1] Alfred V Aho, Ravi Sethi und Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison wesley, 1986.

[2] Boost. *Spirit X3*. 2015. URL: http://boost-spirit.com/.

[3] MongoDB. *MongoDB Documentation*. URL: https://docs.mongodb.com/.

[4] Robert C Moore. "Removing left recursion from context-free grammars". In: *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*. Association for Computational Linguistics. 2000, S. 249–255.

[5] Kian Win Ong, Yannis Papakonstantinou und Romain Vernoux. "The SQL++ Query Language: Configurable, Unifying and Semi-structured". In: *CoRR*. 2014, abs/1405.3631. URL: http://arxiv.org/abs/1405.3631.

[6] PostgreSQL. *The Path of a Query*. URL: https://www.postgresql.org/docs/9.3/static/query-path.html.

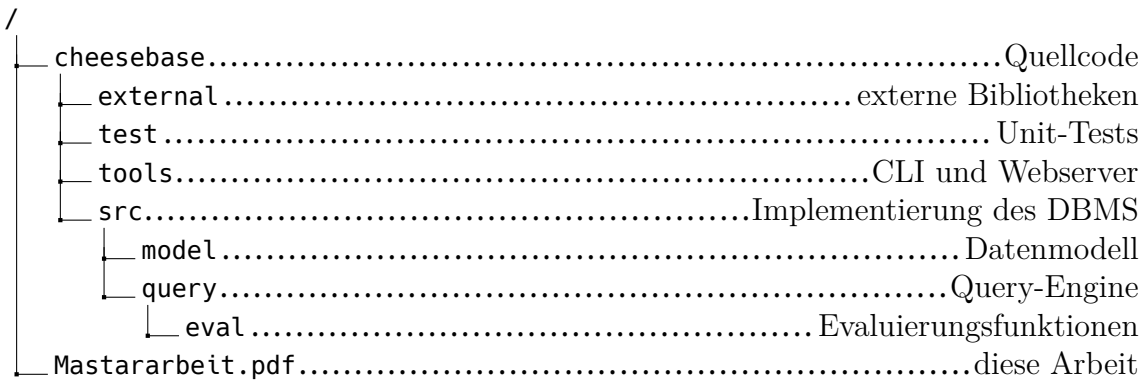
[7] Uwe Schöning. *Theoretische Informatik kurz gefasst*. Wissenschaftsverlag Mannheim, 1992.

[8] SQLite. *Distinctive Features Of SQLite*. URL: http://sqlite.org/different.html.

[9] SQLite. *The SQLite Query Planner*. URL: https://www.sqlite.org/optoverview.html.

[10] Max Taube. *Cheesebase*. 2016. URL: http://github.com/mcheese/cheesebase.

Verzeichnisstruktur der CD



Eidesstatliche Versicherung

Ich erkläre hiermit, dass ich diese Masterarbeit selbstständig ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht. Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum

Unterschrift