Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelorarbeit Informatik

Extract and Sanitize PostgreSQL Query Parse Trees

Peter Richter

31.05.2015

Gutachter / Betreuer

Prof. Dr. Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Richter, Peter:
Extract and Sanitize PostgreSQL Query Parse Trees
Bachelorarbeit Informatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 01.02.2015 - 31.05.2015

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit dem Umwandeln von Parse-Trees aus den Log-Files von PostgreSQL und der Aufbereitung der daraus gewonnen Daten. Das entwickelte Programm ist ein SQL-Parser, der mit Informationen angereicherte SQL-Querys in JSON ausgibt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der Lehrstuhl für Datenbanksysteme an der Universität Tübingen arbeitet an einer Vielzahl verschiedener Forschungsprojekte.

Diese Projekte gehen in viele verschiedene Richtungen, basieren jedoch häufig auf der internen Repräsentation einzelner Querys. Um die verschiedenen Forschungsarbeiten ausführen zu können, wurden schon etliche SQL-Parser entwickelt, welche auf die entsprechenden Projekte abgestimmt waren.

Auch im Datenbanken-System selbst werden diese Querys zu deren Verarbeitung geparst. PostgreSQL nutzt dafür einen ausgeklügelten Parser, in dem bereits Jahre der Entwicklung stecken. Dieser bietet dabei zusätzlich noch eine Vielzahl semantischer Tests, Typinferenz und Vereinfachungen der Querys. Mit den Daten dieses Parsers könnten alle von PostgreSQL lesbaren Querys geparst werden. Die Daten dieses Parsers würden sich daher hervorragend für weitergehende Forschungen an den Querys eignen.

Diese Arbeit hat sich daher damit beschäftigt, diese Daten aus PostgreSQL zu extrahieren. Um an diese Daten zu gelangen wurden die Log-Files genutzt, die vom Datenbankensystem erzeugt werden. Dabei ist es möglich, sich in den Log-Files unter anderem den Parse-Tree ausgeben zu lassen. In diesem sind umfangreiche Informationen enthalten. Leider sind diese Daten in dieser Form noch nicht gut zu verarbeiten. Zu diesem Zweck wurde ein Programm entwickelt, welches daraus eine infomationsreiche und gut strukturierte Form erstellt. Dazu werden die Daten bereinigt, aufgearbeitet und in JSON als universell einsetzbares Format umgewandelt.

Diese Daten können nun für die verschiedensten Projekte genutzt werden. Eine Möglichkeit der Weiterverwendung könnte in der Forschung des Lehrstuhls bei der Analyse der Data-Provenance bestehen. Diese untersucht die Herkunft von Ergebnissen, also auf welchen Daten das Ergebnis der SQL-Query basiert.

Alternativ könnten die Daten auch für die Entwicklung bzw. Optimierung eines SQL Debuggers genutzt werden.

In dieser Arbeit wird zunächst in Kapitel 2 der Aufbau des Programms beschrieben. Nachdem kurz auf die verwendeten Programme und Sprachen eingegangen wurde, wird erläutert, wie PostgreSQL zu konfigurieren ist und der grundlegende Ablauf des Programms beschrieben.

In Kapitel 3 werden die Parse-Trees der Log-Files genauer erklärt. Dies beinhaltet zum einen ihre Grammatik und zum anderen ihre relevanten Strukturen und Attribute.

Da PostgreSQL eine interne Repräsentation zur Darstellung von Werten von Konstanten besitzt, beschäftigt sich Kapitel 4 mit deren Umwandlung.

Das Ausgabeformat wird anhand von Beispielen ausführlich in Kapitel 5 beschrieben.

Abschließend wird in Kapitel 6 auf potentielle Probleme des Programms und mögliche zukünftige Weiterentwicklungen eingegangen.

Kapitel 2

Aufbau des Programms

2.1 Programme, Sprachen und Module

Für die Entwicklung des Programms wurde das freie Datenbanksystem **PostgreSQL** benutzt. Dieses liefert ausführliche Log-Files, welche die Basis des Programms darstellen. Zu Beginn der Arbeit wurde PostgreSQL in der Version 9.3 genutzt. Allerdings wurden bereits während der Implementierung die Anpassungen auf Version 9.4 durchgeführt und daher wird PostgreSQL in dieser Version vorausgesetzt. Ältere Versionen von PostgreSQL können daher ohne Anpassungen des Programms nicht uneingeschränkt genutzt werden.

Als Programmiersprache wurde **Python** in der Version 3.4 verwendet. Diese unterstützt mehrerer Plattformen und die Syntax von Python bietet eine kompakte und übersichtliche Darstellung des Quellcodes.

Zum Parsen der Log-Files wurde **yapps2**[1] benutzt. Yapps2 ist ein Python Parsergenerator der von Amit J. Patel entwickelt wurde. Er ist frei verfügbar (MIT-Lizenz[2]), die Bedienung ist einfach und es lassen daher mit ihm schnell verschiedene Parser erzeugen. Die erzeugten Parser sind nicht sehr schnell, aber für die geplanten Anwendungszwecke dieser Arbeit ausreichend.

Um verschiedene Anfragen aus Python an PostgreSQL stellen zu können wurde **Psycopg**[3] in Version 2.5 verwendet. Dieses ist ebenso frei verfügbar (GNU Lesser General Public License[4]).

Die dargestellten Diagramme, Tabellen und Zeichnungen wurden mit **yEd**, **LaTeX** und **TextMate** erstellt.

[1] http://theory.stanford.edu/~amitp/yapps/
[2] http://opensource.org/licenses/MIT
[3] http://initd.org/psycopg/
[4] http://www.gnu.org/licenses/lgpl-3.0-standalone.html

2.2 PostgreSQL Setup

Bevor PostgreSQL die für dieses Programm notwendigen Log-Files erzeugen kann, müssen einige Einstellungen vorgenommen werden. Dazu werden zunächst in der PostgreSQL Konfigurations-Datei (`postgresql.conf`) der `logging_collector`, `debug_print_parse` und `debug_pritty_print` aktiviert.

Der `logging_collector` und `debug_print_parse` müssen auf **on** und `debug_pritty_print` auf **off** gestellt werden, da bei aktiviertem `debug_pritty_print` fehlerhafte Daten beim Parsen der Log-Files entstehen können.

Daraufhin werden die Parse-Trees aller Anfragen an das Datenbanksystem in einem Log-File gespeichert. In welchen Abständen oder Dateigrößen neue Log-Files erzeugt werden, kann ebenfalls der PostgreSQL Konfigurations-Datei eingestellt werden.

Die so erzeugten Files werden nun von dem Programm eingelesen (vgl. Punkt 2.5).

2.3 Parsen der Log-Files

Um die von PostgreSQL erstellten Log-Files lesen zu können, wurde eine yapps2-Grammatik entworfen um einen Parser generieren zu lassen. Dieser wandelt die Log-Files in entsprechende Python Strukturen um.

Der erste Schritt des Programms ist das Einlesen und Parsen der Log-Files. Dabei wird eine Liste angelegt, in der die einzelnen Parse-Trees gespeichert werden. Für jeden Parse-Tree wird somit in der Liste ein Python-Dictionary mit den entsprechenden Daten des Parse-Trees erstellt.

Nachdem das Log-File vollständig geparst wurde, liegen die einzelnen Parse-Trees in Python als Liste von Dictionaries vor.

Dabei steht jedes dieser Dictionaries für eine einzelne Query, in der jede Struktur (vgl. Kapitel 3) als einzelnes Dictionary gespeichert wurde. Der Name der Struktur wurde unter dem Key `dicType` und die einzelnen Werte der Attribute unter dem jeweiligen Attributnamen abgelegt.

2.4 Aufbereiten der gewonnenen Daten

Das Aufbereiten der Daten findet in `phase2.py` statt. Bevor die nun gewonnenen Daten weiterverwendet werden, werden mit der Methode `seperate` die

Informationen entfernt, welche für den weiteren Verlauf dieser Arbeit nicht benötigt werden.

Danach wird die PostgreSQL-Darstellung von Konstanten mit der Methode `find_pg_value` in ein benutzerfreundliches Format umgewandelt. Die einzelnen Methoden zur Umwandlung finden sich in der Datei `decodePgNumeric.py`.

2.4.1 Extrahieren benötigter PostgreSQL Relationen

Damit alle Informationen aus den Log-Files von PostgreSQL verarbeitet werden können, müssen im Vorfeld einige Informationen aus der Datenbank abgefragt werden. Hierzu werden in `phase2.py` verschiedene Datenbankabfragen gesendet.
Hierfür wird die Relation `pg_operator` mit den Attributen `oid`, `oprname`, `oprleft`, `oprright` und `oprresult` verwendet, um aus den oids beliebiger Operatoren den passenden Namen (`oprname`) und die passenden Datentypen der Argumente zu finden. Diese Informationen werden in einem Dictionary mit dem Namen `pg_operator` gespeichert.

PostgreSQL hat in seinen Logfiles eine eigene Darstellung von Boolschen Operatoren und speichert diese nicht in `pg_operator`. Daher werden dem Dictionary `pg_operator` noch die Boolschen Operatoren `and`, `or` und `not` hinzugefügt.

Die Typinformationen werden in PostgreSQL in der Relation `pg_type` gespeichert. Dazu werden die Attribute `oid, typname` und `typcategory` abgefragt.

Zum Nachschlagen von Funktionen (zum Beispiel Function Calls, Casts, Aggregate oder Window Functions) wird die Relation `pg_proc` verwendet.

Informationen zu Relationen werden aus den beiden Relationen `pg_class` und `pg_attribute` ausgelesen und in einem Dictionary `pg_class` gespeichert.

2.4.2 Erzeugen des Ausgabeformats

Nachdem alle benötigten Daten zusammengetragen und aufbereitet wurden, wird mit der Umwandlung in das Ausgabeformat begonnen. In `phase3.py` wird für jede Query in der Liste ein neues Objekt der Klasse `BuldTree` erzeugt. Die Methode `createTree` der Klasse `BuildTree` erzeugt für die gewünschte Query ein Dictionary im passenden Ausgabeformat, welches mit der Methode `getTree` zurückgegeben wird.

Dieses Dictionary wird nun in JSON umgewandelt und in einer Datei gespeichert.

2.5 Bedienen des Programms

Zum Ausführen des Programms müssen sowohl Python als auch Psycopg installiert werden. Da beim Schritt des Extrahierens der PostgreSQL Relationen spezifische Daten der jeweiligen Datenbank benötigt werden, die die Log-Files erzeugt hat, muss der Zugriff auf dieses Datenbanksystem gewährleistet sein.

2.5.1 config-File

In der Datei `config.py` muss unter dem Variablennamen `db_connect` die entsprechenden Einstellungen für die Verbindung zur Datenbank eingestellt werden. Dazu zählen beispielsweise der Datenbankname, Benutzername und Benutzerpasswort. Die Einstellmöglichkeiten der Variablen entsprechen den Verbindungsparametern (connection parameter des Objekts `connect`) von psycopg2[5]. Unter der Variable `logfile` befindet sich der Pfad des Log-Files, welches umgewandelt werden soll. Die Variable `notWantedKeys` gibt diejenigen Attribute an, welche mit der Methode `seperate` entfernt werden sollen. Die Variable `notWantedDict` löscht ganze Dictionarys.

2.5.2 Programmstart

Um ein Log-File mit dessen Parse-Trees in einzelne Files des gewünschten Ausgangsformats umzuwandeln. Wird nun der Befehl `python phase3.py` ausgeführt. Optional kann nach phase3.py noch der Pfad zum Log-File angegeben werden, um ein anderes Log-File zu parsen als das in `config.py` angegebene. Die ausgegebenen JSON-Files werden dann unter dem im Config-File angegeben Pfad gespeichert.

2.6 Tests

Um das Programm zu testen wurden Beispielquerys aus den diversen .test Files der Suite Sqllogictest[6] extrahiert. Jedes dieser .test Files wurde in ein eigenes SQL-File mit den jeweiligen SQL-Querys umgewandelt. Darauf wurden einige dieser SQL-Files eingelesen und die einzelnen Querys mithilfe von Psycopg in PostgreSQL ausgeführt. Die dabei entstandenen Log-Files wurden nun verwendet um die Tests auszuführen.

Um die Grammatik des Parsers zu testen, wurde ein Unittest implementiert. Dieser hat im Zuge mehrerer Tests ca. 3 Millionen Querys auf Programmab-

[5] http://initd.org/psycopg/docs/module.html
[6] https://github.com/grahn/sqllogictest

brüche während des Parsens überprüft. Außerdem wurde geprüft, ob in der Liste der geparsten Querys jede Query das Attribut `:commandType` enthält.

Aus dem Pool an Beispielquerys haben ca. weitere 300 000 unterschiedliche Querys alle Schritte des Programms durchlaufen und wurden dann in entsprechende JSON Files umgewandelt. Dabei kam es zu keinen Programmabstürzen oder Fehlermeldungen. Die Ausgabe der umgewandelten Querys wurde hierbei nur für einige zufällige Ausgabedateien exemplarisch getestet.

2.7 Beispielanwendung

Um die Möglichkeiten der erzeugten Darstellung zu testen und etwaige Fehler in der Darstellung zu finden, wurde ein Python-Programm entwickelt, welches die erzeugten Daten wieder in valide SQL-Querys umwandelt.

Als erstes wird hierbei ein einzelnes oder mehrere der erzeugten JSON-Files wieder in Python eingelesen. Danach wird für jedes JSON-File ein `BuildQuery`-Objekt angelegt. Beim Anlegen des Objekts werden die eingelesenen Daten in einen String umgewandelt, welcher die Query enthält. Diese Query ist äquivalent zu der ursprünglich geparsten Query. Die Methode `getQuery` der Klasse `BuildQuery` gibt den String mit der SQL-Query zurück.

Es hat sich gezeigt, das sich alle implementierten Strukturen aus den JSON-Files wieder zurück in äquivalente Queries umwandeln lassen.

Kapitel 3

Die PostgreSQL-Log-Files

In diesem Kapitel wird die Bedeutung der von mir verwendeten Strukturen und Attribute der Log-Files dargestellt. Anhand von Beispielen werden alle relevanten Strukturen erläutert.

3.1 Grammatik der Log-Files

Die zum Einlesen der Parse-Trees aus den Log-Files notwendige Grammatik kann wie folgt beschrieben werden:

```
S            ::=  [ \s ]+
char         ::=  <a Unicode character>
dicName      ::=  [A-Z]*
boolean      ::=  'true' | 'false'
null         ::=  '<>'
number       ::=  '-'? S* [0 - 9]+
value        ::=  [\w |\W]+

start        ::=  parsetrees

parsetrees   ::=  ((char+) parsetree)+
parsetree    ::=  'LOG:' S 'parse tree:' S 'DETAIL:' S dict ptend
ptend        ::=  ('CONTEXT:' | 'STATEMENT:')

dict         ::=  '{' dicName S attr+ '}'
attr         ::=  ':' value S attrValue
attrValue    ::=  (number | pgnumeric | boolean |
                  char+ | dict | list | null)

pgnumeric    ::=  number '[' number+ ']'
list         ::=  '(' (attr | boolean | char+ |
                  number | dict | list)* ')'
```

3.2 Relevante Strukturen und deren Attribute

Um mit den Informationen der Log-Files arbeiten zu können ist es wichtig diese auch zu verstehen. In den Header-Dateien `parsenodes.h` und `primnodes.h` des PostgreSQL Source Codes befindet sich die Deklaration der einzelnen Strukturen und Attribute. Aus diesen Dateien habe ich mir die Bedeutung der einzelnen Attribute erarbeitet, um sie dann passend kombinieren zu können.

Dabei verwende ich im Folgenden den Begriff '*Struktur*' für die von PostgreSQL erstellten `structs`. In den Log-Files tauchen diese direkt nach einer öffnenden geschweiften Klammen in Großbuchstaben auf. Darauf folgen die einzelnen Attribute der Struktur. Diese beginnen mit einem Doppelpunkt, worauf direkt der Name des Attributs folgt. Der Wert des Attributs folgt auf den Namen des Attributs.

```
1  {CONST // Eine Struktur mit dem Namen CONST
2          :consttype 23  //Ein Attribut ':consttype' mit dem Wert '23'
3          :consttypmod -1
4          :constcollid 0
5          :constlen 4
6          :constbyval true
7          :constisnull false
8          :location 7
9          :constvalue 4 [ 1 0 0 0 0 0 0 0 ]
10         // Ein Attribut ':constvalue'  mit dem Wert: '4 [ 1 0 0 0 0 0 0 0 ]'
11 }
```

Listing 3.1: Struktur mit Attributen für eine Konstante mit dem Wert 1

Im vorherigen Kapitel wurde im Programmablauf bereits erwähnt, dass die PostgreSQL Darstellung von Konstanten in ein benutzerfreundlicheres Format umgewandelt werden muss. Der Wert der Konstanten ist im oberen Beispiel mit 4 [1 0 0 0 0 0 0 0] angegeben. Dies entspricht dem Wert `1`. Diese Darstellung wird in folgenden Beispielen häufiger auftreten. Wie solche Werte umgewandelt werden, wird im nächsten Kapitel erläutert.

In den folgenden Unterpunkten werden anhand von Beispielen die verwendeten Strukturen und Attribute eingeführt. Dabei werden nur die in dieser Arbeit verwendeten Attribute erklärt, weitere Attribute der einzelnen Strukturen werden dabei bewusst vernachlässigt. Zur übersichtlicheren Darstellung werden die Beispiele der erzeugten Log-Files mit dem Befehl `debug_pritty_print` **on** dargestellt. Desweiteren sind ab Listing 3.3 alle Listings in gekürzter Form dargestellt und enthalten unter Umständen nicht alle Attribute.

Eine der einfachsten Anfragen an das Datenbanksystem wäre:

SELECT 1;

Dieser erzeugt den folgenden Parse-Tree im Log-File:

```
1  LOG:  parse tree:
2  DETAIL:     {QUERY // Struktur Query
3              :commandType 1 // 1 = Select Statement
4              :querySource 0
5              :canSetTag true
6              :utilityStmt <>
7              :resultRelation 0
8              :hasAggs false
9              :hasWindowFuncs false
10             :hasSubLinks false
11             :hasDistinctOn false
12             :hasRecursive false
13             :hasModifyingCTE false
14             :hasForUpdate false
15             :cteList <>
16             :rtable <>
17             :jointree
18                {FROMEXPR
19                :fromlist <>  // Liste der FROM-Ausdruecke
20                :quals <> // Liste der WHERE-Ausdruecke
21                }
22             :targetList ( // Liste der SELECT-Ausdruecke
23                {TARGETENTRY  // Einzelner SELECT-Ausdruck
24                :expr
25                   {CONST // Konstante
26                   :consttype 23
27                   :consttypmod -1
28                   :constcollid 0
29                   :constlen 4
30                   :constbyval true
31                   :constisnull false
32                   :location 7
33                   :constvalue 4 [ 1 0 0 0 0 0 0 0 ]
34                   }
35                :resno 1
36                :resname ?column?
37                :ressortgroupref 0
38                :resorigtbl 0
39                :resorigcol 0
40                :resjunk false
41                }
42             )
43             :withCheckOptions <>
44             :returningList <>
45             :groupClause <>
46             :havingQual <>
47             :windowClause <>
48             :distinctClause <>
49             :sortClause <>
50             :limitOffset <>
51             :limitCount <>
52             :rowMarks <>
53             :setOperations <>
54             :constraintDeps <>
55             }
56 STATEMENT:  SELECT 1;
```

Listing 3.2: Logfile für SELECT 1;

PostgreSQL schreibt *'LOG: parse tree:'* in das Log-File und beginnt nach *'DETAIL:'* den Parse-Tree auszugeben, beginnend mit der Struktur `QUERY` (Zeile 2).

3.2.1 Query

Die Struktur `QUERY` steht für eine Anfrage an das Datenbankensystem. Alle Informationen die für die Ausführung der Anfrage notwendig sind, werden hier gespeichert.

Das Attribut `:commandtype` einer `QUERY` gibt die Art der Anfrage an. Hierbei steht `1` für *SELECT* Querys, `2` für *INSERT* Querys, `3` für *UPDATE* Querys, `4` für *DELETE* Querys oder `5` für *CREATE TABLE* Querys und andere Utility Statements. Dieses Programm beschäftigt sich nur mit *SELECT* Querys und damit auch nur mit `:commandtype 1`.

In `:jointree` findet sich die Struktur `FROMEXPR` mit den einzelnen Ausdrücken der *FROM* und *WHERE* Clause.

Weitere Attribute der Struktur `QUERY` finden sich in dieser Tabelle[1]:

	Attribut	Bedeutung
[0]	`:commandType`	Art der Anfrage (1 = *SELECT* Query)
[5]	`:hasAggs`	boolscher Wert ob Aggregate in der Query vorkommen
[6]	`:hasWindowFuncs`	boolscher Wert ob Window Functions in der Query vorkommen
[7]	`:hasSubLinks`	boolscher Wert ob `SUBLINK`s in der Query vorkommen
[8]	`:hasDistinctOn`	boolscher Wert ob *DISTINCT ON* in der Query vorkommt
[9]	`:hasRecursive`	boolscher Wert ob die Query rekursiv ist
[12]	`:cteList`	Liste von `COMMONTABLEEXPRESSION`s.
[13]	`:rtable`	Liste von `RTE`s welche Informationen zu Relationen, Joins, Values, Funktionen und CTEs enthalten
[14]	`:jointree`	Enthält eine `FROMEXPR` mit den Ausdrücken der *FROM* und *WHERE* Clause
[15]	`:targetList`	Liste mit `TARGETENTRY`s, welche den einzelnen Ausdrücken der *SELECT* Clause entsprechen
[18]	`:groupClause`	Liste von `SORTGROUPCLAUSE`s, welche auf die in *GROUP BY*s festgelegten Spalten referenzieren

[1] Die Zahl in der ersten Spalte gibt in dieser und allen anderen Tabellen die Stelle innerhalb der Struktur an, an der sich das Attribut befindet.

[19]	`:havingQual`	Liste von `SORTGROUPCLAUSE`s, welche auf die in *`HAVING`* festgelegten Spalten referenzieren
[20]	`:windowClause`	Beinhaltet eine Liste entsprechender *`WINDOW`* Clauses mit den entsprechenden Optionen
[21]	`:distinctClause`	Liste von `SORTGROUPCLAUSE`s, welche auf die in *`DISTINCT ON`* festgelegten Spalten referenzieren
[22]	`:sortClause`	Liste von `SORTGROUPCLAUSE`s, welche auf die in *`ORDER BY`* festgelegten Spalten referenzieren
[23]	`:limitOffset`	gesetzter Offset - Wert
[24]	`:limitCount`	gesetzter Limit - Wert
[25]	`:setOperations`	Operation-Tree für *`UNION, INTERSECT, EXCEPT`* Statements

Tabelle 3.1: `QUERY`

3.2.2 TargetEntry

Unter dem Attribut `:targetList` der Struktur `QUERY` ist eine Liste von `TARGETENTRY`s gespeichert. Ein `TARGETENTRY` stellt einen einzelnen Ausdruck der `Select` Clause dar.

In `:expr` ist der exakte Ausdruck gespeichert, für den das `TARGETENTRY` angelegt wurde. Je nach Art des Ausdrucks findet sich im Attribut die passende Expression. Beispiele für Expressions wären `VAR`, `CONST`, `AGGREF` und `WINDOWFUNC`.

Das Attribut `:resname` gibt den Spaltennamen des Attributs an. Falls ein Name im *`SELECT`* Clause für das Attribut vergeben wurde (beispielsweise `SELECT foo AS bar`), findet sich dieser in `:resname`. Anderenfalls wird hier der Spaltenname der Relation übernommen. Falls kein Name existiert, wird von PostgreSQL ein Name generiert (vlg. Listing 3.2 Zeile 36).

Falls ein *`DISTINCT, GROUP BY, HAVING, PARTITION BY`* oder *`ORDER BY`* auf das Attribut zeigt, entspricht `:ressortgroupref` dem Attribut `:tleSortGroupRef` der entsprechenden `SORTGROUPCLAUSE` (Kapitel 3.2.9).

Ist ein `TARGETENTRY` intern nur für die Verarbeitung der Query angelegt worden, soll aber nicht ausgegeben werden, nimmt `:resjunk` den Wert `true` an.

In Beispiel (Listing 3.2) steht in `:expr` eine Konstante (`CONST`) mit dem Wert `1`.

	Attribut	Bedeutung
[0]	`:expr`	Expression auf die sich der `TARGETENTRY` bezieht
[2]	`:resname`	Name des Attributs
[3]	`:ressortgroupref`	Verweis auf ein `SORTGROUPCLAUSE`
[6]	`:resjunk`	Boolscher Wert, ob Eintrag in der Ausgabe angezeigt wird oder nicht

Tabelle 3.2: `TARGETENTRY`

3.2.3 Const

Die Struktur `CONST` ist die Darstellung von Konstanten.

Das Attribut `:consttype` speichert den Datentyp der Kontante als `oid`. Sie kann in PostgreSQL in der Relation `pg_type` nachgeschlagen werden. Die `oid` `23` steht beispielsweise für den Datentyp `int4`.

`:constvalue` speichert den Wert der Konstanten in der PostgreSQL Notation. Die Umwandlung der PostgreSQL Notation in das Ursprungsformat wird in Kapitel 4 erläutert.

	Attribut	Bedeutung
[0]	`:consttype`	`oid` des Datentyps der Konstanten
[7]	`:constvalue`	Wert der Konstanten

Tabelle 3.3: `CONST`

Beispiel

Die weiteren Beispiele beziehen sich nun auf eine Relation ***users***, welche die Spalten ***name, rating*** und ***stars*** besitzt.
name und ***stars*** sind vom Datentyp *text* und ***rating*** vom Typ *integer*.

Für die Query:

```
SELECT name FROM users WHERE rating = 1;
```

ergibt sich folgende `:targetList` im Log-File:

```
...
:targetList (
   {TARGETENTRY
   :expr
      {VAR
      :varno 1 // Index auf :rtable
      :varattno 1 // Spaltennummer der Relation
      :vartype 25 // Datentyp der Spalte
      :vartypmod -1
      :varcollid 100
      :varlevelsup 0
      :varnoold 1
      :varoattno 1
      :location 7
      }
   :resno 1
   :resname name
   ...
   :resjunk false
   }
)
...
```

Listing 3.3: `:targetList` für *`SELECT name FROM users WHERE rating = 1;`*

3.2.4 Var

`VAR` steht für eine Spalte in der Relation.

In `:varattno` findet sich die Attributnummer der Relation, auf welche sich diese `VAR` bezieht, oder 0, falls es auf alle Attribute einer Relation zeigt.

In `:vartype` ist der Datentyp der Spalte gespeichert (`oid` aus `pg_type`).

	Attribut	Bedeutung
[0]	`:varno`	Index auf die zugehörige Relation
[1]	`:varattno`	Index auf das entsprechende Attribut
[2]	`:vartype`	`oid` des Datentyps aus der Relation `pg_type`
[5]	`:varlevelsup`	Angabe, wieviele Ebenen hinaufgestiegen werden müssen, um mithilfe der `:varno` zur passenden `:rtable` zu gelangen

Tabelle 3.4: `VAR`

Den Index (beginnend mit 1) der zugehörigen Relation ist in `:varno` gespeichert. Die zugehörige Relation findet sich im `:rtable` der `QUERY`.

Im Beispiel ist `:varno 1`. Das bedeutet, dass das Element an der Position 0 der Liste in `:rtable` die zugehörige Relation ist. Diese ist als `RTE` gespeichert.

3.2.5 RangeTableEntry (RTE)

Ein `RTE` steht beispielsweise für eine Relation im Datenbankensystem.

Die `:rtable` der vorherigen Beispielquery mit dem `RTE` einer Relation:

```
1            ...
2            :rtable (
3               {RTE
4               :alias <>
5               :eref
6                  {ALIAS
7                  :aliasname users  // Name der Relation
8                  :colnames ("name" "rating" "stars") // Spaltennamen
9                  }
10              :rtekind 0
11              :relid 16385 // oid der Relation
12              :relkind r
13              :lateral false
14              :inh true
15              :inFromCl true
16              :requiredPerms 2
17              :checkAsUser 0
18              :selectedCols (b 9 10)
19              :modifiedCols (b)
20              :securityQuals <>
21              }
22           )
23           ...
```

Listing 3.4: `:rtable` von *`SELECT name FROM users WHERE rating = 1;`*

In `:eref` findet sich ein `ALIAS`. Dieser hat unter `:colnames` eine Liste von Strings gespeichert, die alle Spaltennamen der Relation benennt. Wurden in der `FROM` Clause Spaltennamen umbenannt, finden sich hier die umbenannten Spaltennamen. Der Name der Relation (bzw. der neue umbenannte Name) findet sich im `ALIAS` unter `:aliasname`.

Das Attribut `:relid` gibt die `oid` der Relation an. Diese kann in der PostgreSQL Relation `pg_class` nachgeschlagen werden.

Das Attribut `:rtekind` beschreibt von welcher Art die `RTE` ist. Sie kann für eine Relation[2], eine Subquery, das Ergebnis eines Joins, eine Function, das Konstrukt Values oder eine CommonTableExpression stehen (vgl. Tabelle 3.5 und 3.6a).

Je nachdem von welcher Art die `RTE` ist, steht sie für verschiedene Strukturen und es existieren verschiedene Attribute innerhalb eines `RTE`. In Tabelle 3.5 sind die Attribute des `RTE` aufgeführt und bei welchem `:rtekind` sie auftreten.

[2] Als Relation ist in diesem Zusammenhang jeder Eintrag der Relation `pg_class` zu verstehen. Der Typ des Eintrags ist in `:relkind` gespeichert und entspricht dem Werten von `pg_class.relkind`.

	Kind	Attribut	Bedeutung
[1]	0-5	`:eref`	Hierunter befindet sich ein `ALIAS` mit dem Attribut `:colnames`, welcher die Namen der Spalten angibt
[2]	0-5	`:rtekind`	Kind des `RTE` (0-5)
[9]	0-5	`:lateral`	Boolscher Wert ob der `RTE` lateral ist
[3]	0	`:relid`	`oid` der zugehörigen PostgreSQL Relation aus `pg_class` (vlg. 2.4.1)
[3]	0	`:rekind`	Art der Relation (vlg. `pg_class.relkind`)
[3]	1	`:subquery`	Subquery der Struktur `QUERY`, die in der *`FROM`* Clause definiert wurde
[3]	2	`:jointype`	Art des Joins (0-3) (vlg. Tabelle 3.6b)
[4]	2	`:joinaliasvars`	Liste von `VAR`, welche den Spalten des Join-Ergebnisses entsprechen
[3]	3	`:functions`	Enthält eine `RANGETBLFUNCTION` (vgl. 3.2.18) (seit PostgreSQL 9.4 steht hier eine Liste von `RANGETBLFUNCTION`s)
[3]	4	`:values_lists`	Liste von Listen von Expressions, welche für jeweils eine Row stehen
[3]	5	`:ctename`	Name stimmt mit dem Attribut `:ctename` der entsprechenden `COMMONTABLEEXPR` überein
[6]	5	`:ctecoltypes`	Enthält Informationen der Datentypen der einzelnen Spalten. Dies wird als Liste von `oid`s dargestellt, welche in der PostgreSQL Relation pg_type nachgeschlagen werden können

Tabelle 3.5: `RTE`

	Bedeutung
`:rtekind 0`	Relation
`:rtekind 1`	Subquery
`:rtekind 2`	Join
`:rtekind 3`	Function
`:rtekind 4`	Values
`:rtekind 5`	CTE

(a) `:rtekind`

	Bedeutung
`:jointype 0`	*`INNER JOIN`*
`:jointype 1`	*`LEFT OUTER JOIN`*
`:jointype 2`	*`FULL OUTER JOIN`*
`:jointype 3`	*`RIGHT OUTER JOIN`*

(b) `:jointype`

Tabelle 3.6: `:rtekind` und `:jointype`

3.2.6 Fromexpr

Die `FROMEXPR` enthält alle Informationen der *`FROM`* und der *`WHERE`* Clause.

In `:fromlist` ist die Liste von Ausdrücken, die in der *`FROM`* Clause angefordert wurde, gespeichert.
Jeder dieser Ausdrücke ist ein `RANGETBLREF`. Ein `RANGETBLREF` ist eine Referenz auf eine existierende `RTE`. Der Index (beginnend mit 1) des zugehörigen `RTE` ist in `:rtindex` gespeichert. Dieser ist im `:rtable` der `QUERY` zu finden.

Wurde ein expliziter *`JOIN`* ausgeführt, kann in der `:fromlist` statt einer `RANGETBLREF` auch eine `JOINEXPR` (3.2.8) stehen.

Im Beispiel ist `:rtindex 1`. Das bedeutet, die `RTE` an der Position 0 der Liste in `:rtable` ist die zugehörige Relation *`users`* (vlg. Listing 3.4).

```
1            ...
2            :jointree
3               {FROMEXPR // FROM Clause
4               :fromlist ( // Liste von Argumenten der FROM Clause
5                  {RANGETBLREF
6                  :rtindex 1
7                  }
8               )
9               :quals // WHERE Clause
10                 {OPEXPR  // "Operator-Funktion"
11                 :opno 96  // oid des Operators
12                 :opfuncid 65
13                 :opresulttype 16
14                 :opretset false
15                 :opcollid 0
16                 :inputcollid 0
17                 :args ( // Liste der Funktionsargumente
18                    {VAR // Spalte 'rating'
19                    :varno 1
20                    :varattno 2
21                    :vartype 23
22                    :vartypmod -1
23                    :varcollid 0
24                    :varlevelsup 0
25                    :varnoold 1
26                    :varoattno 2
27                    :location 29
28                    }
29                    {CONST // Konstante mit dem Wert '1'
30                    :consttype 23
31                    :consttypmod -1
32                    :constcollid 0
33                    :constlen 4
34                    :constbyval true
35                    :constisnull false
36                    :location 38
37                    :constvalue 4 [ 1 0 0 0 0 0 0 0 ]
38                    }
39                 )
40                 :location 36
41                 }
42              }
43           ...
```

Listing 3.5: `:jointree` von *`SELECT name FROM users WHERE rating = 1;`*

	Attribut	Bedeutung
[0]	`:fromlist`	Liste von `RANGETBLREF`s und / oder `JOINEXPR`s (siehe 3.2.8)
[1]	`:quals`	Ausdruck der *`WHERE`* Clause

Tabelle 3.7: `FROMEXPR`

In `:quals` steht der Ausdruck der *`WHERE`* Clause. Im Beispiel wäre dies der Vergleich der Werte der Spalte *`rating`* mit der Konstanten `1`. Dieser Vergleich ist als `OPEXPR` dargestellt.

3.2.7 Opexpr

`OPEXPR` stellt eine Funktion dar, die mit Hilfe eines Operators der Relation `pg_operator` ausgeführt wird. Eine `OPEXPR` besitzt Informationen über die `oid` des Operators, die `oid` der darunterliegenden Funktion in `pg_proc`, die Argumente mit denen die Funktion aufgerufen wird und die `oid` des Datentyps des Funktionsergebnisses.

	Attribut	Bedeutung
[0]	`:opno`	`oid` des Operators aus `pg_operator`
[1]	`:opfuncid`	`oid` der Funktion aus `pg_proc`
[2]	`:opresulttype`	`oid` des Datentyps aus `pg_type`
[6]	`:args`	Liste der Argumente des Operators (ein bis zwei Argumente), beispielsweise `VAR` oder `CONST`

Tabelle 3.8: `OPEXPR`

Im Beispiel entspricht der Operator mit der `:opno 96` dem Operator '=‘. Der Datentyp des Ergebnisses `:opresulttype 16` wäre ein *boolean*. Des Weiteren lässt sich in der Liste der Argumente (`:args`) ablesen, dass das erste Argument die Spalte *`rating`* und das zweite Argument die Konstante `1` ist.

Unter der Prämisse, dass die `OPEXPR` im Attribut `:quals` steht, entspricht dies genau dem eingegebenen Ausdruck *`WHERE rating = 1`*.

3.2.8 Joinexpr

In 3.2.6 wurde beschrieben, dass eine `:fromlist` anstatt eines `RANGETBLREF`s auch eine `JOINEXPR` enthalten kann. `JOINEXPR` ist die Darstellung eines Joins.

Eine `JOINEXPR` besitzt die Information von welcher Art ein *`JOIN`* ist, ob es ein *`NATURAL JOIN`* war, welche Relationen am Join beteiligt sind und einen Index auf das Ergebnis des Joins. Dieses Ergebnis findet sich in der zugehörigen `QUERY` unter dem Attribut `:rtable` wieder.

	Attribut	Bedeutung
[0]	`:jointype`	Art des Joins (0-3) (vgl. Tabelle 3.6b)
[1]	`:isNatural`	Boolscher Wert, ob der Join *`NATURAL`* war
[2]	`:larg`	`RANGETBLREF` des linken Arguments des Joins
[3]	`:rarg`	`RANGETBLREF` des rechten Arguments des Joins
[5]	`:quals`	ggf. Bedingungen welche für den Join gelten
[7]	`:rtindex`	Index auf die zugehörige Relation

Tabelle 3.9: `JOINEXPR`

```
1            :jointree
2               {FROMEXPR
3               :fromlist (
4                  {JOINEXPR // JOIN Ausdruck
5                  :jointype 0
6                  :isNatural false
7                  :larg  // linkes Argument
8                     {RANGETBLREF
9                     :rtindex 1
10                    }
11                 :rarg // rechtes Argument
12                    {RANGETBLREF
13                    :rtindex 2
14                    }
15                 :usingClause <>
16                 :quals <> // ON-Bedingungen des Joins
17                 :alias <>
18                 :rtindex 3 // Index des Ergebnises des Joins
19                 }
20              )
21              :quals <> // WHERE Clause
22              }
```

Listing 3.6: `:jointree` von
`SELECT * FROM users AS foo CROSS JOIN users AS bar;`

Beispielsweise ist der *`CROSS JOIN`* in Listing 3.6 ein `INNER JOIN` (da `:jointype 0`). Das linke Argument befindet sich an Index 1, das rechte Argument an Index 2 und das Ergebnis des Joins an Index 3 in `:rtable`.

Alle Joins werden über die in Tabelle 3.6b beschriebenen Jointypen dargestellt. Um andere Joins darstellen zu können werden die entsprechenden *`ON`*-Bedingungen in `:quals` der `JOINEXPR` gesetzt.

3.2.9 SortGroupClause

Eine `SORTGROUPCLAUSE` ist die Repräsentation der SQL-Befehle *`DISTINCT`*, *`GROUP BY`*, *`HAVING`*, *`ORDER BY`* und *`PARTITION BY`* in den Log-Files.

Beispiel : *`SELECT avg(rating) FROM users GROUP BY stars;`*

```
1            :groupClause (
2               {SORTGROUPCLAUSE
3               :tleSortGroupRef 1
4               :eqop 98  // oid des Vergleichsoperators
5               :sortop 664  // oid des Sortieroperators
6               :nulls_first false
7               :hashable true
8               }
9            )
```

Listing 3.7: Beispiel `SORTGROUPCLAUSE` in der `:groupClause`

Ein `SORTGROUPCLAUSE` referenziert mit `:tleSortGroupRef` auf ein `TARGETENTRY`, auf welchem der entsprechende Befehl (*`GROUP BY`*, *`DISTINCT`*, *`ORDER BY`*, *`...`*) ausgeführt wird.

Im Beispiel ist `:tleSortGroupRef 1`. Damit zeigt es auf die Spalte *`stars`*, da in diesem `TARGETENTRY` das Attribut `:ressortgroupref` den selben Wert aufweist (vgl. Listing 3.8). Da der Ausdruck *`stars`* nicht in der Ausgabe erscheinen soll, ist das Attribut `:resjunk true`.

```
1               {TARGETENTRY //Spalte 'stars'
2               :expr
3                  {VAR
4                  :varno 1
5                  :varattno 3
6                  :vartype 25
7                  :vartypmod -1
8                  :varcollid 100
9                  :varlevelsup 0
10                 :varnoold 1
11                 :varoattno 3
12                 :location 39
13                 }
14              :resno 2
15              :resname <>
16              :ressortgroupref 1
17              :resorigtbl 0
18              :resorigcol 0
19              :resjunk true // Spalte tritt im OUTPUT nicht auf
20              }
```

Listing 3.8: Auszug der `:targetList` von *`SELECT avg(rating) FROM users GROUP BY stars;`*

	Attribut	Bedeutung
[0]	`:tleSortGroupRef`	Zahl, welche sich auf `TARGETENTRY`s mit dem gleichen Wert im Attribut `:ressortgroupref` bezieht
[1]	`:eqop`	`oid` des Vergleichsoperators
[2]	`:sortop`	`oid` des Sortieroperators
[3]	`:nulls_first`	boolscher Wert, ob *`NULL`* Values vor anderen Werten kommt

Tabelle 3.10: `SORTGROUPCLAUSE`

3.2.10 Aggref

`AGGREF` steht für eine Aggregatfunktion.

Beispiel eines (gekürzten) `TARGETENTRY`s einer `AGGREF`.

```
1   {TARGETENTRY
2   :expr
3      {AGGREF  // Aggregat
4      :aggfnoid 2101 // oid der Aggregatfunktion
5      :aggtype 1700 // oid des Ergebnisdatentyps
6      :aggcollid 0
7      :inputcollid 0
8      :aggdirectargs <>
9      :args (
10        {TARGETENTRY  // Spalte 'rating'
11        :expr
12           {VAR
13           :varno 1
14           :varattno 2
15           :vartype 23
16           [...]
17           }
18        :resno 1
19        [...]
20        }
21     )
22     :aggorder <> // ORDER BY Clause im Aggregat
23     :aggdistinct <> // DISTINCT Clause im Aggregat
24     :aggfilter <> // FILTER Clause im Aggregat
25     :aggstar false
26     :aggvariadic false
27     :aggkind n
28     :agglevelsup 0
29     :location 7
30     }
31  :resno 1
32  :resname avg // Name der ausgegebenen Spalte
33  :ressortgroupref 0
34  :resorigtbl 0
35  :resorigcol 0
36  :resjunk false
37  }...
```

Listing 3.9: Auszug der `:targetList` von *`SELECT avg(rating) FROM users GROUP BY stars;`*

Die AGGREF enthält die oid der dahinter liegenden Funktion der PostgreSQL Relation pg_proc, die oid des Datentyps des Ergebnisses, die Argumente auf denen das Aggregat wirkt und potentielle *ORDER BY, DISTINCT* und *FILTER*, welche auf dem Aggregat wirken.

	Attribut	Bedeutung
[0]	:aggfnoid	oid der Funktion aus der Relation pg_proc
[1]	:aggtype	oid des Ergebnisdatentyps aus pg_type
[5]	:args	Liste von Argumenten, auf die das Aggregat wirkt
[6]	:aggorder	Liste von SORTGROUPCLAUSES für *ORDER BY*s im Aggregat
[7]	:aggdistinct	Liste von SORTGROUPCLAUSES für *DISTINCT*s im Aggregat
[8]	:aggfilter	FILTER, der auf das Aggregat wirkt

Tabelle 3.11: AGGREF

3.2.11 Windowfunc

Eine WINDOWFUNC steht für Window-Functions und ist ähnlich aufgebaut wie eine AGGREF (vgl. Tabelle 3.11). Beispielsweise wird :aggfnoid zu :winfnoid. Der Name des Attributs :aggfilter bleibt allerdings gleich und bestimmte Attribute wie :aggdistinct oder :aggorder kommen nicht vor.

Zusätzlich besitzt eine WINDOWFUNC das Attribut :winref, welches auf die zu ihr zugehörige WINDOWCLAUSE zeigt. In dieser steht die *OVER* Clause der Window-Functions.

	Attribut	Bedeutung
[0]	:winfnoid	oid der Funktion aus der Relation pg_proc
[1]	:wintype	oid des Datentyps aus der Relation pg_type
[4]	:args	Liste von Expressions, welche die Funktion übergeben bekommen hat
[5]	:aggfilter	Filter, der auf der Window-Function wirkt
[6]	:winref	Wenn die Zahl in diesem Attribut mit dem Wert des Attributs :winref der Struktur WINDOWCLAUSE übereinstimmt, gehören diese Optionen zu ihr

Tabelle 3.12: WINDOWFUNC

3.2.12 Windowclause

Die `WINDOWCLAUSE` gibt an, welche Optionen für die referenzierte `WINDOWFUNC` in der *`OVER`* Clause gesetzt sind. Sie tritt immer dann auf, wenn eine Window-Function in der Query benutzt wurde.

	Attribut	Bedeutung
[2]	`:partitionClause`	Liste von `SORTGROUPCLAUSE`s nach denen partitioniert wird
[3]	`:orderClause`	Liste von `SORTGROUPCLAUSE`s nach denen sortiert wird
[4]	`:frameOptions`	Zahl in der die Windowoptions codiert sind (vgl. Tabelle 3.14)
[5]	`:startOffset`	Expression, die den Startwert angibt
[6]	`:endOffset`	Expression, die den Endwert angibt
[7]	`:winref`	Wenn die Zahl in diesem Attribut mit dem Wert des Attributs `:winref` der Struktur `WINDOWFUNC` übereinstimmt, gehören diese Optionen zu ihr

Tabelle 3.13: `WINDOWCLAUSE`

Im Attribut `:frameOptions` sind weitere Optionen gespeichert. Wenn man aus dem dort gesetzten Wert das Bitmuster extrahiert und dann die gesetzen Bits mit denen aus Tabelle 3.14 vergleicht, erhält man die gesetzten Optionen.

Angenommen wir hätten die folgende Query:

`SELECT avg(rating) OVER (RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM users;`

Dann würde die `WINDOWCLAUSE` wie folgt aussehen:

```
1            :windowClause (
2               {WINDOWCLAUSE
3               :name <>
4               :refname <>
5               :partitionClause <>
6               :orderClause <>
7               :frameOptions 539
8               :startOffset <>
9               :endOffset <>
10              :winref 1
11              :copiedOrder false
12              }
13           )
```

Listing 3.10: Auszug der `:windowClause` von *`SELECT avg(rating) OVER (RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM users;`*

In diesem Beispiel besitzt `:frameOptions` den Wert 539 (1000011011_2). Die Optionen `RANGE, BETWEEN, START UNBOUNDED PRECEDING` und `END CURRENT ROW` wären folglich gesetzt (vgl. Abb. 3.1). Wenn Optionen gesetzt werden ist auch die Option `NONDEFAULT` immer gesetzt.

Die Option `BETWEEN` wird nur dann gesetzt, wenn es in der Query explizit geschrieben wurde. Falls nur die Startgrenze gesetzt ist (ohne *`BETWEEN`* in der SQL-Query), wird die Endgrenze von PostgreSQL selbst gesetzt, aber die Option `BETWEEN` ist nicht aktiv.

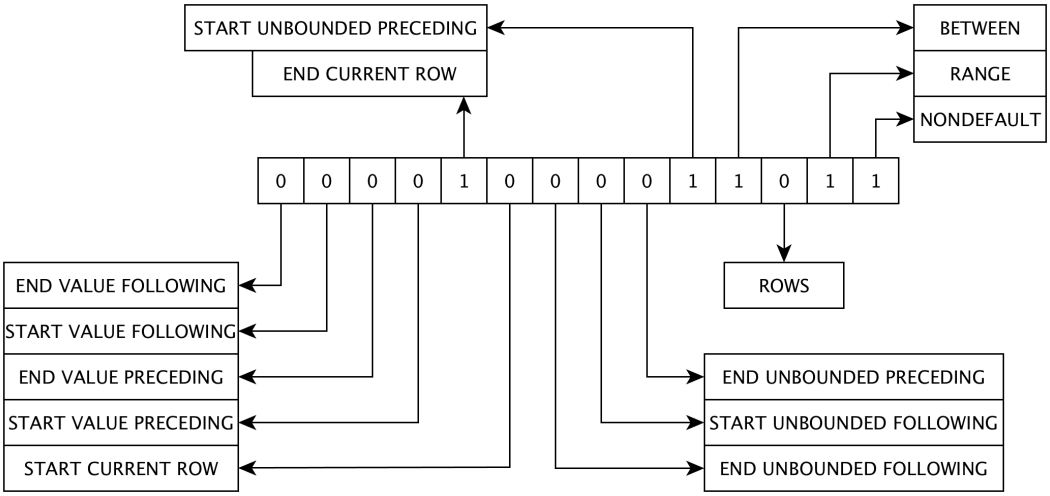

Abbildung 3.1: Beispiel `:frameOptions 539`

Bitmuster	Bedeutung
0x00001	`NONDEFAULT?` (Sind Optionen gesetzt?)
0x00002	`RANGE` Verhalten?
0x00004	`ROWS` Verhalten?
0x00008	`BETWEEN` gegeben?
0x00010	Start ist `UNBOUNDED PRECEDING`
0x00020	Ende ist `UNBOUNDED PRECEDING` (nicht erlaubt)
0x00040	Start ist `UNBOUNDED FOLLOWING` (nicht erlaubt)
0x00080	Ende ist `UNBOUNDED FOLLOWING`
0x00100	Start ist `CURRENT ROW`
0x00200	Ende ist `CURRENT ROW`
0x00400	Start ist `VALUE PRECEDING`
0x00800	Ende ist `VALUE PRECEDING`
0x01000	Start ist `VALUE FOLLOWING`
0x02000	Ende ist `VALUE FOLLOWING`

Tabelle 3.14: `:frameOptions`

3.2.13 Sublink

Subquerys, welche nicht innerhalb einer *FROM* Clause vorkommen, werden als SUBLINK dargestellt (also nicht als RTE). Es wird nach der Art des Sublinks unterschieden.

Für die Ausdrücke *EXISTS, ANY, ALL* und *ARRAY* mit passender Subquery, wird ein entsprechender Sublink vom Typ EXISTS_SUBLINK, ALL_SUBLINK, ANY_SUBLINK und *ARRAY_SUBLINK* angelegt. Es existieren außerdem noch EXPR_SUBLINK, ROWCOMPARE_SUBLINK und CTE_SUBLINK.

Als EXPR_SUBLINK gelten Subquerys (auserhalb der *FROM* Clause) mit einer einzelnen Spalte als Rückgabewert.

ROWCOMPARE_SUBLINKs kommen beispielsweise beim Vergleich mehrerer Spalten mit einer Subquery vor (vgl. Listing 3.16). Soll nur eine Spalte verglichen werden, so wird dies mittels eines EXPR_SUBLINKs realisiert.

CTE_SUBLINKs tauchen nur in der Struktur SUBPLAN[3] für CommonTableExpressions auf, nicht aber in SUBLINKs.

	Attribut	Bedeutung
[0]	:subLinkType	vgl. Tabelle 3.16
[1]	:testexpr	Testausdruck der geprüft wird
[3]	:subselect	Subquery (QUERY) für die der Sublink erzeugt wurde

Tabelle 3.15: SUBLINK

In :testexpr findet sich bei :subLinkType 0 und 4-6 der Wert NULL statt des Testausdrucks.

	Bedeutung
:subLinkType 0	EXISTS_SUBLINK
:subLinkType 1	ALL_SUBLINK
:subLinkType 2	ANY_SUBLINK
:subLinkType 3	ROWCOMPARE_SUBLINK
:subLinkType 4	EXPR_SUBLINK
:subLinkType 5	ARRAY_SUBLINK
:subLinkType 6	CTE_SUBLINK

Tabelle 3.16: :subLinkType

[3] Auf die Struktur SUBPLAN wird hier nicht weiter eingegangen, da sie für diese Arbeit keine Relevanz hat

3.2.14 Boolexpr

In Listing 3.11 findet sich unter dem Attribut `:testexpr` eine `BOOLEXPR`. Eine `BOOLEXPR` steht für eine Funktion mit einem der Boolschen Operatoren `AND`, `OR` oder `NOT`.

In `:boolop` steht der Operator (`and`, `or` oder `not`) und in `:args` die Argumente der Funktion. Der Operator `NOT` besitzt genau ein Argument und die Operatoren `AND` und `OR` zwei oder mehr Argumente.

Für die gegebene Query stehen als Argumente die zwei `OPEXPR`s, welche die Spalten *`users.rating`* mit *`foo.rating`* und *`users.stars`* mit *`foo.stars`* vergleichen.

Hierbei wird *`foo.rating`* und *`foo.stars`* als `PARAM` (siehe 3.2.15) dargestellt.

```
1   {SUBLINK
2   :subLinkType 3
3   :testexpr
4      {BOOLEXPR // Funktion welche and, or oder not als Operator besitzt
5      :boolop and  // Operator and
6      :args (   // Liste der Argumente der BOOLEXPR
7         {OPEXPR
8         :opno 96 // Operator '='
9         [...]
10        :args (
11           {VAR
12           :varno 1
13           :varattno 2 // Spalte 'rating' der Query
14           :vartype 23
15           [...]
16           }
17           {PARAM // Referenz auf eine Ausgabespalte 'rating' des Subselects
18           :paramkind 2
19           :paramid 1 // Index auf die Ausgabe des :subselect
20           :paramtype 23 // oid des Datentyps der Spalte
21           [...]
22           }
23        )
24        :location 42
25        }
26        [...]
27        )
28  [...]
```

Listing 3.11: gekürzter Auszug der `:testexpr` des `SUBLINK`s von *`SELECT * FROM users WHERE (rating, stars) = (SELECT foo.rating, foo.stars FROM users as foo WHERE name = stars);`*

3.2.15 Param

Ein PARAM vom :paramkind 2 steht für eine Spalte der Ausgabe eines Subselects. Die anderen Arten des Parameters sind für diese Arbeit nicht relevant.

Ein PARAM besitzt das Attribut :paramid, dessen Index auf das passende TARGETENTRY in der :targetList des :subselects zeigt.

:paramid 1 zeigt also auf das erste Element in :targetList der QUERY in :subselect.

	Attribut	Bedeutung
[0]	:paramkind 2	PARAM_SUBLINK
[1]	:paramid	Index des Parameters
[3]	:paramtype	oid des Datentyps

Tabelle 3.17: PARAM

3.2.16 Setoperationstmt

SETOPERATIONSTMT ist die PostgreSQL Darstellung der Mengenoperationen wie *UNION, INTERSECT* und *EXCEPT*.

:op gibt die Art der Mengenoperation an. Dabei steht 1 für ein *UNION*, 2 für ein *INTERSECT* und 3 für ein *EXCEPT*.

Die Argumente der Operation finden sich in :larg und :rarg. Die Mengenoperationen werden mit Hilfe der Liste von SORTGROUPCLAUSES des Attributs :groupClauses ausgeführt. Bei *UNION ALL* ist der Wert dieses Attributs NULL.

	Attribut	Bedeutung
[0]	:op	Art des SETOPERATIONSTMT
[1]	:all	Boolscher Wert, ob *ALL* gesetzt war oder nicht
[2]	:larg	RANGETBLREF für das linke Argument
[3]	:rarg	RANGETBLREF für das rechte Argument
[4]	:colTypes	Liste von oids (von Datentypen der Relation pg_type) der Ausgabe der Operation
[7]	:groupClauses	Eine Liste von SORTGROUPCLAUSES oder NULL

Tabelle 3.18: SETOPERATIONSTMT

```
:setOperations
   {SETOPERATIONSTMT
   :op 1  // UNION Operator
   :all false  //ALL nicht gesetzt
   :larg
      {RANGETBLREF // linkes Argument
      :rtindex 1
      }
   :rarg
      {RANGETBLREF //rechtes Argument
      :rtindex 2
      }
   :colTypes (o 25)
   :colTypmods (i -1)
   :colCollations (o 100)
   :groupClauses (
      {SORTGROUPCLAUSE // entfernt hier Duplikate aus dem Ergebnis
      :tleSortGroupRef 0
      :eqop 98
      :sortop 664
      :nulls_first false
      :hashable true
      }
   )
}
```

Listing 3.12: Auszug der `:setOperations` von
`SELECT name FROM users UNION SELECT stars FROM users;`

3.2.17 CommonTableExpr

`COMMONTABLEEXPR` ist die Darstellung einer einzelnen Common Table Expression (innerhalb der `:cteList` der `QUERY`).

Sie besitzt einen Namen, die Datentypen und Namen der einzelnen Spalten und die Subquery, für welche `COMMONTABLEEXPR` steht.

Das Attribut `:cterecursive` gibt an, ob diese Query rekursiv ist, während Attribut `:hasRecursive` der Struktur `QUERY` angibt, ob in der `:cteList` irgendeine der `COMMONTABLEEXPR` rekursiv ist.

	Attribut	Bedeutung
[0]	`:ctename`	Name der CTE
[2]	`:ctequery`	Subquery für welche die CTE steht
[4]	`:cterecursive`	Boolscher Wert, ob diese Subquery rekursiv ist
[6]	`:ctecolnames`	Liste von Namen der einzelnen Spalten
[7]	`:ctecoltypes`	Liste von `oid`s der Datentypen der Spalten

Tabelle 3.19: `COMMONTABLEEXPR`

Im Parse-Tree der Beispielquery

*WITH test AS (SELECT * FROM users) SELECT * FROM test, users;*

ist in dem Attribut `:rtable` ein `RTE` vom Kind 5 zu finden, welcher auf die `COMMONTABLEEXPR` mit dem `:ctename` *test* in der `:cteList` verweist. Das Attribut `:ctecoltypes` des `RTE` und das der `COMMONTABLEEXPR` besitzen die gleichen Werte.

```
[...]
  :cteList (
     {COMMONTABLEEXPR
     :ctename test
     :aliascolnames <>
     :ctequery
        {QUERY
        [...]
        }
     :location 5
     :cterecursive false
     :cterefcount 1
     :ctecolnames ("name" "rating" "stars")
     :ctecoltypes (o 25 23 25)
     :ctecoltypmods (i -1 -1 -1)
     :ctecolcollations (o 100 0 100)
     }
  )
  :rtable (
     {RTE
     :alias <>
     :eref
        {ALIAS
        :aliasname test
        :colnames ("name" "rating" "stars")
        }
     :rtekind 5
     :ctename test
     :ctelevelsup 0
     :self_reference false
     :ctecoltypes (o 25 23 25)
     :ctecoltypmods (i -1 -1 -1)
     :ctecolcollations (o 100 0 100)
     :lateral false
     :inh false
     :inFromCl true
     :requiredPerms 2
     :checkAsUser 0
     :selectedCols (b)
     :modifiedCols (b)
     :securityQuals <>
     }
     [...]
```

Listing 3.13: gekürzter Auszug der `:cteList` von *WITH test AS (SELECT * FROM users) SELECT * FROM test, users;*

3.2.18 Rangetblfunction

Unter Punkt 3.2.5 wurden verschiedene Arten von `RTE`s dargestellt. Ein Funktions-`RTE` besitzt den `:rtekind 3`. Das Attribut `:functions` ist eine Liste von `RANGETBLFUNCTION`s. In einer `RANGETBLFUNCTION` ist eine Funktion innerhalb des `RTE` gespeichert. Im Attribut `:funcexpr` ist dessen Expression-Tree von Funktionsaufrufen als `FUNCEXPR` gespeichert.

Ein Beispiel dafür wäre:

*SELECT * FROM sin(1);*

```
:rtable (
   {RTE
   :alias <>
   :eref
      {ALIAS
      :aliasname sin
      :colnames ("sin")
      }
   :rtekind 3 // Funktions-RTE
   :functions ( // Liste der Funktionen
      {RANGETBLFUNCTION
      :funcexpr
         {FUNCEXPR //Die Funktion der RANGETBLFUNCTION
         :funcid 1604 // oid der Funktion -> sin
         :funcresulttype 701
         :funcretset false
         :funcvariadic false
         :funcformat 0 // 0 = EXPLICIT_CALL
         :funccollid 0
         :inputcollid 0
         :args ( // Argumente der Funktion
            {FUNCEXPR
            :funcid 316  // oid der Funktion -> float8
            :funcresulttype 701
            :funcretset false
            :funcvariadic false
            :funcformat 2 // 2 = IMPLICIT_CAST
            :funccollid 0
            :inputcollid 0
            :args (
               {CONST
               :consttype 23
               [...]
               }
            )
         [...]
      }
   )
   [...]
   }
)
```

Listing 3.14: gekürzter Auszug der `:rtable` von *SELECT * FROM sin(1);*

Im Beispiel ist im Attribut `:funcexpr` der `RANGETBLFUNCTION` die Funktion *sin* gespeichert, welche als Argument wieder eine Funktion hält, die implizit die Zahl `1` von einem *int4* in ein *float8* castet.

3.2.19 Funcexpr

`FUNCEXPR` steht für eine PostgreSQL Funktion der Relation `pg_proc`.

Sie enthält die `oid` der Funktion, die `oid` des Datentyps des Ergebnisses, die Funktionsargumente und die Art der Funktion. Die Art der Funktion ist in `:funcformat` gespeichert und kann den Wert 0-2 annehmen. Hierbei steht 0 (`EXPLICIT_CALL`) für Funktionsaufrufe und 1 (`EXPLICIT_CAST`) bzw. 2 (`IMPLICIT_CAST`) für Casts.

	Key	Bedeutung
[0]	`:funcid`	`oid` der Funktion aus `pg_proc`
[1]	`:funcresulttype`	`oid` des Ergebnisdatentyps aus `pg_type`
[4]	`:funcformat`	Zahl, welche für die Art der Funktion steht (0-2)
[7]	`:args`	Liste der Argumente der Funktion, beziehungsweise *`NULL`*, falls die Funktion keine Argumente benötigt

Tabelle 3.20: `FUNCEXPR`

Der Cast im vorherigen Beispiel wandelte einen *int4* in ein *float8* um. Dieser wurde als `FUNCEXPR` dargestellt. Manche Casts werden allerdings auch durch andere Strukturen dargestellt. Im Folgenden werden daher die beiden Strukturen `COERCEVIAIO` und `RELABELTYPE` vorgestellt.

3.2.20 CoerceViaIO

Als `COERCEVIAIO` werden Casts dargestellt, deren textuelle Darstellungen gleich sind. Intern ist die Typumwandlung so implementiert, dass der textuelle Output des zu castenden Typs als Eingabe zum Erzeugen des Zieldatentyps fungiert.

	Key	Bedeutung
[0]	`:arg`	Expression die gecastet werden soll
[1]	`:resulttype`	`oid` des Ergebnisdatentyps aus `pg_type`
[3]	`:coerceformat`	Zahl, welche für die Art der Funktion steht (entspricht `:funcformat` einer `FUNCEXPR`)

Tabelle 3.21: `COERCEVIAIO`

3.2.21 RelabelType

`RELABELTYPE` steht für Casts zwischen zwei Datentypen, welche die gleiche binäre Darstellung haben.

Beispielsweise ist die Umwandlung einer `oid` in *int4* ein solcher Cast.

	Key	Bedeutung
[0]	`:arg`	Expression die gecastet werden soll
[1]	`:resulttype`	`oid` des Ergebnisdatentyps aus `pg_type`
[4]	`:relabelformat`	Zahl, welche für die Art der Funktion steht (entspricht `:funcformat` einer `FUNCEXPR`)

Tabelle 3.22: `RELABELTYPE`

3.2.22 Case

Folgendes Beispiel:

`SELECT CASE 42 WHEN 2 THEN 1 WHEN 4 THEN 2 ELSE 0 END;`

Der SQL-Befehl *`CASE`* wird im Parse-Tree mit Hilfe der Struktur `CASE` dargestellt.

Hierbei ist `:casetype` die `oid` des Rückgabedatentyps.

In `:arg` steht der Testausdruck auf den geprüft wird, im Beispiel `42`.

`:args` ist eine Liste von *`WHEN [...] THEN [...]`* Clauses. Diese werden als `WHEN` dargestellt.

Das Attribut `:defresult` gibt die *`ELSE`* Clause an.

	Key	Bedeutung
[0]	`:casetype`	`oid` des Ergebnisdatentyps aus `pg_type`
[2]	`:arg`	Expression gegen die getestet werden soll, beispielsweise `CONST, VAR`
[3]	`:args`	Liste von `WHEN`-Argumenten
[4]	`:defresult`	Expression, welche das Defaultergebnis ist

Tabelle 3.23: `CASE`

```
:targetList (
 {TARGETENTRY
 :expr
   {CASE
   :casetype 23 //oid des Ergebnisdatentyps
   :casecollid 0
   :arg          // Testausdruck
      {CONST
      :consttype 23
      :constvalue 4 [ 42 0 0 0 0 0 0 0 ] // Wert 42 des Testausdrucks
      }
   :args ( // Liste von WHEN Clauses
      {WHEN  // Erste WHEN Clause
      :expr  // Ausdruck der ersten WHEN Clause
         {OPEXPR // Vergleichsfunktion
         :opno 96  // Operator '='
         :opfuncid 65
         :opresulttype 16
         :args (
            {CASETESTEXPR  // Referenz auf den Testausdruck des CASE
            :typeId 23
            }
            {CONST       // Wert gegen den der Testausdruck geprueft wird
            :consttype 23
            :constvalue 4 [ 2 0 0 0 0 0 0 0 ] // Wert 2
            }
         )
         }
      :result // THEN Clause der ersten WHEN Clause
         {CONST
         :consttype 23
         :constvalue 4 [ 1 0 0 0 0 0 0 0 ] // Wert 1
         }
      }
   )
   :defresult // ELSE Clause des CASE Ausdrucks
      {CONST
      :consttype 23
      :constvalue 4 [ 0 0 0 0 0 0 0 0 ] // Wert 0
      }
   }
 :resno 1
 :resname case
 :resjunk false
 }
)
```

Listing 3.15: gekürzter Auszug der `:targetList` von *`SELECT CASE 42 WHEN 2 THEN 1 WHEN 4 THEN 2 ELSE 0 END;`*

3.2.23 When

`WHEN` ist ein Argument eines `CASE`-Ausdrucks auf das getestet wird.

Sie besitzt das Attribut `:expr` in der sich die Funktion befindet, die getestet wird. Dabei ist der Ausdruck aus der CASE Clause, gegen den getestet wird, als `CASTETESTEXPR` dargestellt. Dieser zeigt auf den Ausdruck des Attributs `:arg` der Struktur `CASE`.

In `:result` befindet sich der Ausdruck, der sich in dem *THEN* Clause befindet.

	Key	Bedeutung
[0]	`:expr`	Vergleich des *WHEN*-Ausdrucks
[2]	`:result`	Ergebnis des *WHEN*-Ausdrucks (*THEN*)

Tabelle 3.24: `WHEN`

3.2.24 Casetestexpr

`CASETESTEXPR` ist eine Referenz auf den Testausdruck `:arg` der Struktur `CASE`.

Im Attribut `:typeId` ist die `oid` des Datentyp des Testausdrucks gespeichert.

3.2.25 Coalesce

Die sytaktische Kurzform eines *CASE* Ausdrucks ist *COALESCE*. Diese besitzt eine eigene Struktur mit dem Namen `COALESCE`.

Hierbei ist in `:coalescetype` die `oid` des Ergebnisdatentyps gespeichert. Im Gegensatz zu `CASE` sind in `:args` allerdings keine `WHEN`s gespeichert, sondern es findet sich direkt die Liste von Ausdrücken gegen die getestet wird. (vgl. Listing 3.16)

	Key	Bedeutung
[0]	`:coalescetype`	`oid` des Ergebnisdatentyps aus `pg_type`
[2]	`:args`	Liste von Argumenten auf die *COALESCE* wirkt

Tabelle 3.25: `COALESCE`

```
1   {COALESCE
2   :coalescetype 23 // oid des Rueckgabedatentyps
3   :coalescecollid 0
4   :args ( // Liste von Ausdruecken gegen die getestet wird
5      {CONST // Erster Ausdruck
6      :consttype 23
7      :constvalue 4 [ 1 0 0 0 0 0 0 0 ] // Wert 1
8      }
9      [...] // Weitere Ausdruecke
10     }
11  )
12  }
```

Listing 3.16: gekürzter Auszug einer `COALESCE` der Query *`SELECT COALESCE (1,2,3);`*

Kapitel 4

Umwandeln der PostgreSQL-Darstellung von Konstanten

PostgreSQL verwendet für die Darstellung von Konstanten ein eigenes Format, welches zuerst in ein benutzerfreundliches Format umgewandelt werden muss. Dabei gibt es für viele der Datentypen ein eigenes Format. Die Methoden zur Umwandlung der einzelnen Werte findet sich in der Datei `decodePgNumeric.py`.

Im vorherigen Kapitel ist die Darstellung in verschiedenen Beispielen bereits erschienen. In den folgenden Punkten wird nun die Umwandlung einiger Datentypen erklärt.

4.1 int4

Der Wert 42 beispielsweise wird in PostgreSQL mit 4 [42 0 0 0 0 0 0] dargestellt, der Datentyp dieser Zahl ist *int4* (`oid` = 23).

Im Attribut `:constvalue` steht eine Zahl, gefolgt von einer Liste von Zahlen. In der Liste wird der Wert der jeweiligen Konstanten gespeichert.

Bei einem *int4* werden nur die ersten vier Zahlen aus der Liste verwendet. Die Liste lautet demnach: [42 0 0 0].

Dann müssen die Zahlen der Liste in die Zweierkomplementdarstellung umgewandelt werden. Jede der Zahlen ist ein signed 8-Bit Block. Damit ergibt sich: [00101010 00000000 00000000 00000000]

Wenn die ersten vier Blöcke von Little-Endian in Big-Endian umwandelt werden ergibt sich: [00000000 00000000 00000000 00101010].

Interpretiert man diese vier Blöcke nun als eine Zahl und wandelt diese wieder in eine Dezimalzahl um, erhält man den gespeicherten Wert 42.

Ein weiteres Beispiel: Die Zahl -234 wird als 4 [22 −1 −1 −1 0 0 0 0] dargestellt.

- Es werden nur die ersten 4 Zahlen verwendet:

  [22 −1 −1 −1]

- Zahlen in Zweierkomplementdarstellung umwandeln:

  [00010110 11111111 11111111 11111111]

- Little-Endian in Big-Endian Format:

  [11111111 11111111 11111111 00010110]

- Zahlen zusammenfügen:

  11111111111111111111111100010110

- Zweierkomplementzahl in Dezimalzahl umwandeln:

  11111111111111111111111100010110 = -234

Zahlen vom Datentyp *int8* (`oid` = 20) werden auf die selbe Weise umgewandelt, nur das die ersten acht statt der ersten vier Zahlen verwendet werden, bei *int2* entsprechend die ersten zwei Zahlen.

4.2 bool

Die Umwandlung eines *bool*s (`oid` = 16) erfolgt ähnlich.

Beispiel: Der Wert true wird als 1 [1 0 0 0 0 0 0 0] dargestellt.

- Es wird nur die erste Zahl verwendet:

  [1]

- Zahl in die Zweierkomplementdarstellung umwandeln:

  [00000001]

- Little-Endian in Big-Endian Format:

  [00000001]

- Zweierkomplementzahl in Boolschen Wert umwandeln:

  00000001 = 1 = true

4.3 char

Ein *char* (`oid` = 18) wird auf die selbe Weise wie ein *bool* umgewandelt. Allerdings wird die Zahl am Ende gemäß ASCII-Tabelle durch den passenden Char ersetzt.

4.4 text / bpchar

text(`oid` = 25) und *bpchar* (`oid` = 1042) werden beide gleich umgewandelt. Der Wert 'asdf' wird als 8 [32 0 0 0 97 115 100 102] dargestellt. Die ersten vier Zahlen geben bei dieser Darstellung einen Header mit der Länge der Liste an[1], darauf folgt der Wert der darin gespeichert ist. Um den Wert auszulesen, müssen die einzelnen Zahlen der Reihe nach in der ASCII-Tabelle nachgeschlagen und durch den passenden Char ersetzt werden.
Umwandlung von 8 [32 0 0 0 97 115 100 102]:

- Um die Länge der Liste zu berechnen, wird der 32-Bit Header in Zweierkomplementdarstellung von Little-Endian in Big-Endian umgewandelt:

  [00000000 00000000 00000000 00100000]

  - darauf werden die Zahlen zusammengefügt:

    00000000000000000000000000100000

  - danach wird ein Shift um zwei Bits nach rechts ausgeführt:

    00000000000000000000000000001000 = Länge der Liste = 8 Byte

- Die restlichen 4 Byte der Liste nach dem entfernen des Headers:

  97 115 100 102

- die einzelnen Zahlen in der ASCII-Tabelle nachgeschlagen:

  97 115 100 102 = 'asdf'

[1] Da die Eingabe von variabler Länge ist, kann damit die Länge der Liste berechnet werden. Bei den vorherigen Darstellungen war die Länge der Liste fix.

4.5 numeric

Der Beispielwert 42.24 des Datentyps *numeric* (`oid = 1700`) wird als 10 [40 0 0 0 0 −127 42 0 96 9] dargestellt.

1. Die einzelnen Zahlen innerhalb der Liste werden in Zweierkomplementdarstellung umgewandelt:

   [00101000 00000000 00000000 00000000 00000000 10000001 00101010 00000000 01100000 00001001]

2. Nun werden diese von Little-Endian in Big-Endian umgewandelt und in einzelne Blöcke unterteilt:

   (a) 32-Bit Header Wort

   (b) 16-Bit Flag Wort

   (c) Liste von 16-Bit Wörtern.

3. Das 32-Bit Header Wort speichert die Länge der Liste. Man erhält die Länge indem man den Header um zwei Bits nach rechts shiftet.

   In unserem Beispiel:

   00000000000000000000000000101000 >> 2 = Länge der Liste = 10 Byte

4. 16-Bit Flag Wort (vgl. Abbildung 4.1)

   Die auf den Header folgenden zwei Byte ergeben das 16-Bit Flag Wort

   1000 0001 0000 0000

   Hierbei gibt das erste Bit an, ob der Wert als `NumericShort` oder `NumericLong` gespeichert wurde.

   Das zweite Bit gibt an, ob der gespeicherte Wert eine Zahl ist oder nicht.

   Die folgenden Bits hängen davon ab ob, der Wert in `NumericShort` oder `NumericLong` gespeichert wurde.

   - Im `NumericShort` Format gibt das dritte Bit an, ob die Zahl positiv oder negativ ist. Die folgenden sechs Bits geben die Anzahl der Nachkommastellen (scale) und die letzten sieben Bits das Gewicht der ersten Ziffer an (weight).
   - Im `NumericLong` Format geben die folgenden 14 Bits die Anzahl der Nachkommastellen an (scale). Um das Gewicht (weight) der ersten Ziffer anzugeben, werden hier weitere zwei Byte herangezogen.

Im Flag-Word des Beispiels ist folglich Folgendes codiert:

(a) `NumericShort` Format (da erstes Bit = 1)

(b) Es ist eine Zahl gespeichert (da zweites Bit = 0)

(c) Das Vorzeichen der Zahl ist positiv (da drittes Bit = 0)

(d) Es gibt zwei Nachkommastellen (da scale = 2)

(e) Das Gewicht ist 1 (da weight = 0 und damit $nbase^0 = 10000^0 = 1$)

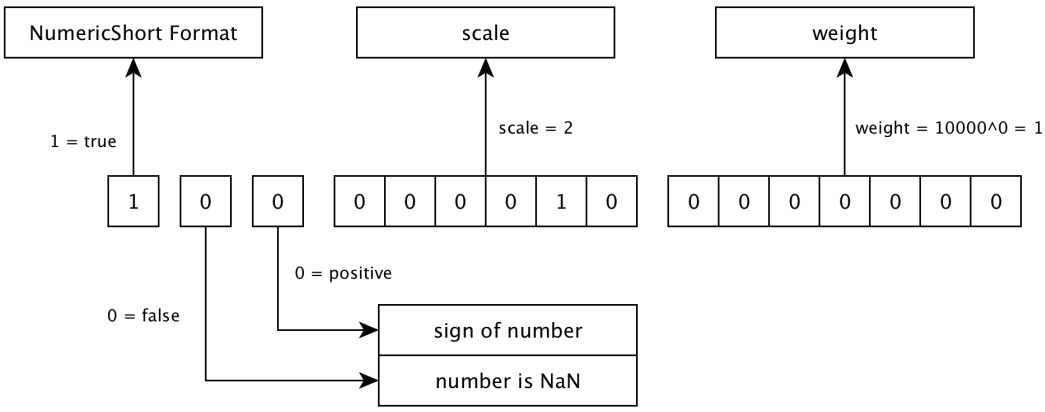

Abbildung 4.1: 16-Bit Flag Word

5. Nun folgen 16-Bit Wörter, die die einzelnen Dezimalstellen enthalten.

   Im Beispiel finden sich hier 2 Wörter:

   (a) $0000000000101010 = 42$

   (b) $0000100101100000 = 2400$

   Werden nun die einzelnen Dezimaldarstellungen der Wörter hintereinander geschrieben und die Nullen am Ende abgeschnitten, erhält man den Wert 4224. Dieser Wert muss noch mit 10000^{Weight} multipliziert werden: $4224 * 10000^0 = 4224$

6. Nun muss das Komma noch an die richtige Position gesetzt werden. Da der scale = 2 ist muss der Wert 4224 mit $10^{-scale} = 10^{-2}$ multipliziert werden.

   $4224 * 10^{-2} = 42.24$

   Damit ist der ursprünglich eingegebene Wert wiederhergestellt.

Kapitel 5

Das Ausgabeformat

In diesem Kapitel wird das Ausgabeformat der erzeugten JSON-Daten beschrieben. Dazu wird zunächst auf die Darstellung der einzelnen Datentypen eingegangen. Danach werden anhand von Beispielen die einzelnen Strukturen des Ausgabeformats erläutert.

5.1 Primitive Datentypen

Im Ausgangsformat gibt es fünf verschiedene Datentypen: `atom`, `array`, `user`, `pseudo` und `unknown`. Diese Datentypen wurden nach der Typkategorie der PostgreSQL Datentypen gegliedert.

Jeder dieser primitiven Datentypen hat die Attribute `typename` und `typecategory`.

Hierbei ist `typename` die genaue Bezeichnung des Datentyps wie beispielsweise *int4*, *int8*, *varchar* oder *text*.

`typecategory` ist die von PostgreSQL vergebene Typkategorie, diese könnte zum Beispiel `S` für *sting types* oder `N` für *numeric types* lauten.

5.1.1 Atom

Als `atom` werden Datentypen verschiedener Typkategorien bezeichnet. Zu `atom` zählen beispielsweise alle Zahldarstellungen, wie *integer* oder *numeric*, aber auch textuelle Darstellungen wie *name*, *varchar* oder *text*. Eine Liste der verschiedenen Typkategorien, welche als `atom` zu finden sind ist in Tabelle 5.1 dargestellt.

5.1.2 Array

Als `array` werden die Datentypen der PostgreSQL Typkategorie Array bezeichnet. Die Namen dieser Datentypen werden von PostgreSQL gewöhnlich beginnend mit einem Unterstrich dargestellt. Der Unterstrich im Namen fällt in dieser Darstellung weg. So hat das Attribut `typename` beispielsweise `int4` statt `_int4` als Wert.

5.1.3 User

Als `user` werden alle Datentypen bezeichnet deren Typkategorie `User-defined types` ist. Beispiele hierfür wären *bytea*, *xid* oder *json*.

5.1.4 Pseudo

`pseudo` steht für Datentypen wie *record*, *any*, *void* oder *opaque*.

5.1.5 Unknown

`unknown` steht für den Datentyp *unknown*.

Datentyp	Code	Typkategorie
`array`	A	Array types
`atom`	B	Boolean types
`composite`	C	Composite types
`atom`	D	Date/time types
`atom`	E	Enum types
`atom`	G	Geometric types
`atom`	I	Network address types
`atom`	N	Numeric types
`pseudo`	P	Pseudo-types
`atom`	R	Range types
`atom`	S	String types
`atom`	T	Timespan types
`user`	U	User-defined types
`atom`	V	Bit-string types
`unknown`	X	unknown type

Tabelle 5.1: Typkategorie

Die Informationen zu `typename` und `typecategory` werden mit Hilfe der `oid` aus der PostgreSQL Relation `pg_type` geholt.

Die `oid` des Datentyps findet sich beispielsweise für die Struktur `CONST` in `:consttype` (vgl Punkt 3.2.3).

```
1 "atom": {
2           "typename": "int4",
3           "typecategory": "N"
4         }
```

(a) JSON-Darstellung

```
atom
 ├ typname text
 └ typcategory S
```

(b) Schematische Darstellung

Abbildung 5.1: Darstellung eines `atom`

Zur übersichtlicheren Darstellung wird im Folgenden die schematische Darstellung bevorzugt verwendet.

5.2 Zusammengesetzte Datentypen

Zusammengesetzten Datentypen können aus primitiven Datentypen und bzw. oder anderen zusammengesetzten Datentypen bestehen. Es gibt vier Arten von zusammengesetzten Datentypen: `col, row, fn` und `bag`.

5.2.1 Column

Der Datentyp einer Spalte einer Relation wird als `col` bezeichnet. Dieser besitzt einen Namen (`name`) und einen Typ (`type`). Der Name ist mit dem der entsprechenden Spalte im Datenbankensystem identisch und wird über die Attributnummer und der `oid` aus der Relation `pg_attribute` gewonnen. Der Typ kann einer der zuvor beschriebenen primitiven Datentypen sein.

```
col
 ├ name rating
 └ type atom
         ├ typname int4
         └ typcategory N
```

Abbildung 5.2: `col` Beispiel

5.2.2 Row

Eine `row` ist der Datentyp einer Liste von Spalten (`col`) und besitzt keine weiteren Attribute.

```
row [ col,                    col,                     col]
      ├ name name              ├ name rating            ├ attname stars
      └ type atom              └ type atom              └ type atom
              ├ typname text           ├ typname int4           ├ typname text
              └ typcategory S          └ typcategory N          └ typcategory S
```

Abbildung 5.3: `row` Beispiel

5.2.3 Function

`fn` ist der Datentyp einer Funktion. Dieser hat die Datentypen der jeweiligen Eingabeargumente als geordnete Liste unter `args` und den Datentyp des Rückgabewerts unter `opres` gespeichert.

```
fn
├ args [atom ,              atom]
|         ├ typename int4    ├ typename int4
|         └ typecategory N   └ typecategory N
|
└ opres atom
          ├ typename bool
          └ typecategory B
```

Abbildung 5.4: `fn` Beispiel

5.2.4 Bag

`bag` steht als Datentyp für eine Menge an Zeilen einer Spalte. Dieser Datentyp ist wichtig für die Darstellung von Aggregatfunktionen oder *`GROUP BY`* Clauses.

```
bag
 └ atom
     ├ typename int4
     └ typecategory N
```

Abbildung 5.5: `bag` Beispiel

5.3 Relevante Strukturen und deren Attribute

Im Folgenden werden verschiedene Strukturen erläutert und zum besseren Verständnis anhand von Beispielen dargestellt.

Als erstes Beispiel dient erneut die Query:

SELECT 1;

```
block
   └ select
      ├ targets [ target ]
      |              ├ name ?column?
      |              └ expr const
      |                        ├ value 1
      |                        └ type atom
      ├ agg false                      ├ typename int4
      └ distinct false                 └ typecategory N
```

Abbildung 5.6: Beispielausgabe der Query *SELECT 1;*

5.3.1 Block

Ein `block` steht für die gesamte Query. Unter einem `block` steht immer eine Struktur `select`, welches für den *SELECT* Teil einer Query steht. Optional werden die Strukturen

`from`, `where`, `groupby`, `having`, `orderby`, `offset` und `limit`

angezeigt, welche die entsprechenden Teile einer Query beschreiben.

Subquerys werden entsprechend auch als `block` dargestellt.
In `offset` und `limit` stehen die gesetzten *OFFSET* bzw. *LIMIT* Werte.

5.3.2 Select

`select` besitzt drei Attribute: `distinct, agg` und `targets`.

Das Attribut `distinct` kann die Werte `true` oder `false` annehmen. Nur falls in der SQL-Query ein *DISTINCT ON* geschrieben wurde, hat `distinct` als Wert eine Liste von `colrefs` (`colref` siehe Punkt 5.3.5).

Das Attribut `agg` kann die Werte `true` oder `false` annehmen, je nachdem ob in `targets` ein Aggregat enthalten ist (vgl. `QUERY↪:hasAggs`).

`targets` ist eine Liste, bei dem jedes Element (`target`) für je einen angefragten Parameter steht (vgl. `QUERY↪:targetList`).

5.3.3 Target

Ein `target` steht für einen angefragten Parameter. `target` besitzt die Attribute `name` und `expr`.

In `name` steht der Spaltenname der Expression. Dieser wird aus der PostgreSQL Struktur `TARGETENTRY`↪`:resname` bezogen (vgl. Punkt 3.2.2 `:resname`).

In `expr` steht der angefragte Parameter.

Im vorherigen Beispiel befindet sich in `target` unter `name` der Wert `?column?` und als `expr` eine Konstante `const`.

5.3.4 Const

Konstanten werden mit der Struktur `const` dargestellt. Ein `const` hat einen Typ (`type`) und einen Wert (`value`).

Der Wert (`value`) der Konstanten wird aus dem Attribut `:constvalue` der Struktur `CONST` übernommen.

Der Typ (`type`) wird mit Hilfe der `oid` in `:constvalue` der Struktur `CONST` erzeugt.

In Abbildung 5.6 beispielsweise hat `const` den Wert 1 und atomaren Datentyp *int4*.

Beispiel

Die weiteren Beispiele beziehen sich nun wieder auf die Relation *`users`*.

Für die Query:

`SELECT bar AS baz FROM users AS foo(bar) Where rating = 1;`

besitzt `SELECT` ↪ `targets` die folgende Struktur:

```
select
 ├ targets [ target ]
 |             ├ name baz
 |             └ expr colref
 |                       ├ row A1
 |                       ├ col bar @1
 |                       └ type atom
 ├ agg false                     ├ typename text
 └ distinct false                └ typecategory S
```

Abbildung 5.7: Beispielausgabe der Query *`SELECT bar AS baz FROM users AS foo(bar) Where rating = 1;`*

5.3.5 Colref

Eine Referenz auf eine Spalte wird mit `colref` bezeichnet. Dies entspricht der PostgreSQL Struktur `VAR`.

Sie besitzt das Attribut `row`, welches die `colref` genau einer `range` zuordnet (mehr zu `range` unter Punkt 5.3.7).

Das Attribut `col` enthält mehrere Informationen. Zum einen gibt es der Spalte einen Namen, der aus `RTE↪:eref↪ALIAS↪:colnames` ausgelesen wird. Zum anderen gibt ein `@<int>` die Position der Spalte innerhalb der Relation an.

Das Attribut `type` beinhaltet einen der zuvor besprochenen Datentypen (siehe Punkt 5.1).

In Abbildung 5.7 ist zu sehen, dass sich `target↪name` und der Name in `colref↪col` unterscheiden.
`target↪name` gibt den ausgegebenen Namen der Spalte der *SELECT*-Clause an, `colref↪col` den Namen der Spalte aus der *FROM*-Clause.

Attribut	Bedeutung
`row`	Name der zugehörigen `range`
`col`	Name der Spalte und ihre Position innerhalb der Relation
`type`	Datentyp der Spalte

Tabelle 5.2: `colref`

5.3.6 From

Die Struktur `from` steht für die *FROM*-Clause einer Query (vgl. `QUERY↪:rtable`).

Sie besitzt das Attribut `ranges`. Dieses ist eine Liste, bei dem jedes Element (`range`) für je einen Ausdruck der *FROM*-Clause steht.

5.3.7 Range

Eine `range` steht für einen Ausdruck der *FROM*-Clause (vgl. `RTE`). Eine `range` besitzt die Attribute: `row, type, lateral` und `expr`.

`row` ist ein Name, welches die Relation eindeutig identifiziert. Dieser wird beginnend bei A1 automatisch generiert und immer weiter hochgezählt (A1, A2, A3...).

`lateral` kann den Wert `true` und `false` annehmen und gibt an, ob die `expr` lateral ist oder nicht. Dieser Wert stammt aus dem Attribut `RTE↪:lateral`.

In `expr` findet sich der Ausdruck dieser `range` aus der *`FROM`*-Clause (zum Beispiel `table`). `type` gibt den Datentyp der `expr` an.

5.3.8 Table

Ein `table` stellt eine Relation im Datenbankensystem dar. Im Attribut `name` wird der Name der Relation dargestellt. Das Attribut `type` ist vom Typ `row` und enthält die Typen der einzelnen Spalten der Relation. Diese Informationen wird über die `oid` in der Relation `pg_class` und `pg_attribute` gewonnen.

Falls in der SQL-Query im *`FROM`*-Teil Spaltennamen umbenannt wurden, werden diese im `type` unter den entsprechenden Namensfeldern unter `col` angezeigt. Der ursprüngliche Spaltenname steht in `expr` in dem entsprechenden Namensfeld.
Im Beispiel findet sich die Relation mit dem Namen *`users`* und den drei Spalten *`name, rating`* und *`stars`*. Zu beachten ist, dass die erste Spalte unter `range↪type↪row ↪col ↪name` mit 'bar' benannt ist, also dem umbenannten Spaltenname. Unter `range ↪expr ↪table↪ type↪ row↪ col ↪name` steht hingegen der ursprüngliche Name der Relation, nämlich 'name'.

5.3.9 Where

`where` beschreibt die *`WHERE`*-Clause einer Query (vlg. `QUERY↪ :jointree↪ :quals`). `where` besitzt das Attribut `expr`, in welcher die *`WHERE`* Bedingung aufgeführt wird. Im Beispiel ist das ein `function call`.

5.3.10 Function Calls

Als `function call` werden alle in PostgreSQL möglichen Funktionen, Operatoren und Casts dargestellt. Dazu zählen beispielsweise die PostgreSQL Strukturen `FUNCEXPR`, `OPEXPR`, `BOOLEXPR`, `SETOPERATIONSTMT` oder `COERCEVIAIO`. Eine Unterscheidung dieser findet nur noch über das Attribut `format` statt.

Ein `function call` besitzt die Attribute `kind, args, type` und `format`.

Dem Attribut `kind` sind mehrere Informationen zu entnehmen. Zum einen woher die Funktion stammt, zum anderen steht, getrennt mit einem Punkt, der Funktions- bzw. Operatorname in diesem Attribut.
`pg_proc.sin` würde also bedeuten, dass die Funktion den Namen `sin` hat und aus der Relation `pg_proc` ist.

```
block
  ├ select
  │  ├ targets [ target ]
  │  │            ├ name baz
  │  │            └ expr colref
  │  │                     ├ row A1
  │  │                     ├ col bar @1
  │  │                     └ type atom
  │  ├ agg false                   ├ typename text
  │  └ distinct false              └ typecategory S
  ├ from
  │   └ ranges [ range ]
  │                 │
  │                 ├ row A1
  │                 ├ type row [ col, ... ]
  │                 │            ├ name bar
  │                 │            └ type atom
  │                 │                    ├ typename text
  │                 │                    └ typecategory S
  │                 ├ lateral false
  │                 └ expr table
  │                          ├ name test
  │                          └ type row [ col, ... ]
  │                                       ├ name name
  │                                       └ type atom
  │                                               ├ typename text
  │                                               └ typecategory S
  └ where
      └ expr function call
                 ├ kind pg_operator.=
                 ├ args [ colref  ,            const ]
                 │          ├ row A1              ├ value 1
                 │          ├ col rating @2       └ type atom
                 │          └ type atom                  ├ typename int4
                 │                  ├ typename int4      └ typecategory N
                 │                  └ typecategory N
                 ├ type fn
                 │    ├ args [atom ,            atom]
                 │    │        ├ typename int4    ├ typename int4
                 │    │        └ typecategory N   └ typecategory n
                 │    │
                 │    └ opres atom
                 │             ├ typename bool
                 │             └ typecategory B
                 └ format FUNCTION_CALL_OP/BOOL
```

Abbildung 5.8: Beispielausgabe von *SELECT bar AS baz FROM users AS foo(bar) Where rating = 1;*

Vor dem Punkt findet sich entweder die PostgreSQL Relation `pg_proc` für Funktionen (`FUNCEXPR`), die PostgreSQL Relation `pg_operator` für Operatoren (`OPEXPR`), `bool` für boolsche Operatoren (`BOOLEXPR`), `io` für bestimmte Casts (`RELABELTYPE` oder `COERCEVIAIO`) oder `statement` für Mengenoperatoren (`SETOPERATIONSTMT`) .

In `args` finden sich die einzelnen Expressions, auf welchen die Funktion operiert. Dies ist eine geordnete Liste der Eingangsparameter.

`type` gibt den Datentyp der Funktion an (`fn`).

Das Attribut `format` unterscheidet die Art der Funktion. Diese kann die folgenden Werte annehmen:

1. `FUNCTION_CALL` für Funktionen
2. `IMPLICIT_CAST / EXPLICIT_CAST` für implizite / explizite Casts
3. `FUNCTION_CALL_OP/BOOL` für Operatoren aus der Relation `pg_operator` bzw. Boolsche Operatoren wie `and, or` oder `not`
4. `FUNCTION_CALL_SETOPERATIONSTMT` für Mengenoperationen

Im Beispiel findet sich im Function Call der Operator `=`, welcher die Werte der Spalte ***rating*** mit der Konstanten `1` vergleicht.

Attribut	Bedeutung
`kind`	Funktionsname und deren Herkunft
`args`	Liste von Expressions, welche die Funktionsargumente darstellen
`type`	Datentyp der Funktion
`format`	Art der Funktion

Tabelle 5.3: `function call`

5.3.11 Join

Ein Join wird durch die Struktur `join` dargestellt (vgl. `JOINEXPR`). Dieser besitzt fünf Attribute: `kind, joinedrow, lhs, rhs` und `pred`.

Der `kind` eines Joins beschreibt dessen Art. Diese kann wieder die Werte `INNER JOIN`, `LEFT OUTER JOIN`, `RIGHT OUTER JOIN` oder `FULL OUTER JOIN` annehmen (vgl. Punkt 3.2.8 `:jointype`).

Die `joinedrow` beschreibt die vom Join erzeugte Relation.

Im Attribut `lhs` steht eine `range`, welche das linke Attribut des Joins, in `rhs` entsprechend eine `range`, welche das rechte Attribut des Joins beschreibt.

Im Attribut `pred` stehen die potentiellen Prädikate eines Joins (vlg. `JOINEXPR↪:quals`).

```
└ expr  join
        ├ kind inner
        ├ joinedrow [ colref ,          ...    colref ]
        |               ├ row A1                ├ row A2
        |               ├ col name @1           ├ col stars @3
        |               └ type atom             └ type atom
        |                       ├ typename text          ├ typename text
        |                       └ typecategory S         └ typecategory S
        ├ lhs range
        |       ├ row A1
        |       ├ type row [col,         ...        col]
        |       |            ├ name name             ├ attname stars
        |       |            └ type atom             └ type atom
        |       |                    ├ typname text          ├ typname text
        |       |                    └ typcategory S         └ typcategory S
        |       ├ lateral false
        |       └ expr table
        |               ├ name users
        |               └ type row [col,     ...         col]
        |                            ├ name name          ├ name stars
        |                            └ type atom          └ type atom
        |                                    ├ typname text        ├ typname text
        |                                    └ typcategory S       └ typcategory S
        |
        ├ rhs range
        |       ├ row A2
        |       ├ type row [col,      ...         col]
        |       |            ├ name name          ├ name stars
        |       |            └ type atom          └ type atom
        |       |                    ├ typname text        ├ typname text
        |       |                    └ typcategory S       └ typcategory S
        |       ├ lateral false
        |       └ expr table
        |               ├ name users
        |               └ type: row [col,        ...     col]
        |                            ├ name name          ├ name stars
        |                            └ type atom          └ type atom
        |                                    ├ typname text       ├ typname text
        |                                    └ typcategory S      └ typcategory S
        └ pred NULL
```

Abbildung 5.9: Auszug der `from` Struktur der Query *`SELECT * FROM users AS foo CROSS JOIN users AS bar;`*

5.3.12 Order By

`orderby` wird als Liste von Kriterien (`crit`) dargestellt (vlg. `SORTGROUCLAUSE`). Jedes dieser Kriterien besitzt drei Attribute:

`asc` kann die Werte `true` oder `false` annehmen, je nachdem ob die Daten auf- oder absteigend sortiert wurden. Dies wird anhand des `:sortop` der `SORTGROUCLAUSE` entschieden.

In `expr` steht ein Ausdruck nach dem die Ausgabe sortiert werden soll.

`nulls_first` kann die Werte `true` oder `false` annehmen, je nachdem ob `NULL` als erstes oder letztes ausgegeben werden soll.

In *`SELECT name FROM users ORDER BY name, rating;`* wird beispielsweise nach zwei Kriterien sortiert, somit hat diese Liste zwei Einträge. Der Eintrag erste steht für *`name`* und der zweite für *`rating`*.

```
└ orderby [ crit,                          crit ]
              ├ asc true                     ├ asc true
              ├ nulls_first false            ├ nulls_first false
              └ expr colref                  └ expr colref
                        ├ row A1                       ├ row A1
                        ├ col name @ 1                 ├ col rating @ 2
                        └ type atom                    └ type atom
                                 ├ typname text                 ├ typname text
                                 └ typcategory S                └ typcategory S
```

Abbildung 5.10: `orderby` Auszug aus *`SELECT name FROM users ORDER BY name, rating;`*

5.3.13 Group By

Beispiel: *`SELECT avg(rating) FROM users GROUP BY stars;`*

Das Attribut `groupby` stellt den Group by - Block einer Query dar. Als Wert besitzt er eine Liste von Expressions. Im Beispiel findet sich die `colref` der Spalte *`stars`* in `groupby`.

```
groupby [ colref ]
              ├ row A1
              ├ col stars @ 3
              └ type atom
                       ├ typname text
                       └ typcategory S
```

Abbildung 5.11: `groupby` Auszug aus *`SELECT avg(rating) FROM users GROUP BY stars;`*

5.3.14 Having

Um das SQL-Konstrukt *HAVING* darzustellen gibt es unter `block` das Attribut `having`, welches genau ein Attribut `expr` besitzt, in dem die Expression des *HAVING*-Befehls steht (vgl. `where` in Punkt 5.3.9). Die Informationen werden aus dem Attribut `:havingQual` der Struktur `QUERY` gewonnen.

5.3.15 Aggregate

Im vorherigen Beispiel wurde die Aggregatfunktion *avg(rating)* in der *SELECT*-Clause aufgerufen. Dies führt zu folgender Ausgabe:

```
select
 ├ targets [ ...,  target ]
 |                   ├ name avg
 |                   └ expr agg
 |                           ├ kind pg_proc.avg
 |                           ├ distinct false
 |                           ├ orderby []
 |                           ├ filter NULL
 |                           ├ type: fn
 |                           |        ├ args [ bag ]
 |                           |        |          └ atom
 |                           |        |             ├ typename int4
 |                           |        |             └ typecategory N
 |                           |        └ opres atom
 |                           |                 ├ typename numeric
 |                           |                 └ typecategory N
 |                           └ args: [ colref ]
 |                                     ├ row A1
 |                                     ├ col rating @2
 |                                     └ type atom
 ├ agg true                                     ├ typename int4
 └ distinct false                               └ typecategory N
```

Abbildung 5.12: `target` Auszug aus *SELECT avg(rating) FROM users GROUP BY stars;*

Aggregate besitzen sechs Attribute: `kind`, `distinct`, `orderby`, `filter`, `type` und `args`.

Im Attribut `kind` findet sich nach der Angabe der Relation `pg_proc.` der Aggregatname. Im Beispiel ist dies `pg_proc.avg` für das *avg* Aggregat.

`distinct` kann die Werte `true` bzw. `false` annehmen, je nachdem ob im Aggregat ein *DISTINCT* gesetzt wurde oder nicht.

In `args` findet sich die Liste von Expressions auf denen die Aggregatfunktion operiert.

Im Attribut `filter` stehen potentielle Filter die die Argumente des Aggregats beeinflussen. Falls kein Filter angegeben ist wird das Attribut wie im Beispiel auf `NULL` gesetzt.

`orderby` enthält die einzelnen Suchkriterien, falls innerhalb des Aggregats ein *`ORDER BY`* aufgetreten ist. Ansonsten enthält `orderby` eine leere Liste.

Attribut	Herkunft	Bedeutung
`kind`	`:aggfnoid`	Aggregatname und dessen Herkunft
`distinct`	`:aggdistinct`	Boolscher Wert, ob *`DISTINCT`* im Aggregat vorkommt
`orderby`	`:aggorder`	Liste von Sortierkriterien auf dem Aggregat
`filter`	`:aggfilter`	Filterfunktion im Aggregat
`type`	`:aggtype`	Datentyp des Aggregats
`args`	`:args`	Argumente auf denen das Aggregat wirkt

Tabelle 5.4: `agg`

5.3.16 Window

`window` wird für die Darstellung von Window Functions verwendet. Dieses besitzt vier Attribute: `function, orderby, partitionby` und `options`.

In `function` befindet sich die Window-Function.

`orderby` enthält die einzelnen Suchkriterien, falls innerhalb der Window-Function ein *`ORDER BY`* aufgetreten ist. Ansonsten enthält `orderby` eine leere Liste (vgl. `WINDOWCLAUSE`↪`:orderClause`).

`partitionby` ist eine Liste von Expressions für die gesetzten Partitions.

In `options` stehen die gesetzten `Frame Clauses` der Window-Function. `options` besitzt fünf Attribute (vgl. `WINDOWCLAUSE`↪`:frameOptions`):

- `start`

  In `start` können die Werte *`UNBOUNDED PRECEDING, UNBOUNDED FOLLOWING, CURRENT ROW, VALUE PRECEDING`* und *`VALUE FOLLOWING`* für den Startwert der `Frame Clause` gesetzt werden.

- `end`

  In `end` können die Werte *`UNBOUNDED PRECEDING, UNBOUNDED FOLLOWING, CURRENT ROW, VALUE PRECEDING`* und *`VALUE FOLLOWING`* für den Endwert der `Frame Clause` gesetzt werden.

- `behavior`

  `behavior` gibt an, ob in der Window-Function *ROWS* oder *RANGE* gesetzt war.

- `start_value`

  Falls das Attribut `start` mit *VALUE PRECEDING/FOLLOWING* gesetzt war, steht hier der der gesetzte Wert. In allen anderen Fällen ist dieses Attribut `NULL`.

- `end_value`

  Falls das Attribut `end` mit *VALUE PRECEDING/FOLLOWING* gesetzt war, steht hier der der gesetzte Wert. In allen anderen Fällen ist dieses Attribut `NULL`.

```
select
 ├ targets [ target ]
 |            ├ name avg
 |            └ expr window
 |                    ├ function agg
 |                    |            ├ kind pg_proc.avg
 |                    |            ├ distinct false
 |                    |            ├ orderby []
 |                    |            ├ filter NULL
 |                    |            ├ type  fn
 |                    |            |        ├ args [ bag ]
 |                    |            |        |          └ atom
 |                    |            |        |             ├ typename int4
 |                    |            |        |             └ typecategory N
 |                    |            |        └ opres atom
 |                    |            |                 ├ typename numeric
 |                    |            |                 └ typecategory N
 |                    |            └ args [ colref ]
 |                    |                       ├ row A1
 |                    |                       ├ col rating @2
 |                    |                       └ type atom
 |                    |                               ├ typename int4
 |                    ├ orderby []                      └ typecategory N
 |                    ├ partitionby []
 |                    └ options
 |                         ├ behavior RANGE
 |                         ├ start UNBOUNDED PRECEDING
 |                         ├ end CURRENT ROW
 |                         ├ start_value NULL
 |                         └ end_value NULL
 ├ agg false
 └ distinct false
```

Abbildung 5.13: Auszug aus *SELECT avg(rating) OVER (RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM users;*

Falls keine Option oder nicht alle Attribute gesetzt werden, werden die übrigen Attribute auf ihren Defaultwert gesetzt.

Die Default Werte sind:

```
1                   "options": {
2                           "start": "UNBOUNDED PRECEDING",
3                           "end_value": null,
4                           "end": "CURRENT ROW",
5                           "behavior": "RANGE",
6                           "start_value": null
7                       }
```

Abbildung 5.14: Default Werte für die `options` einer Window-Function

5.3.17 Sublink

`sublink` besitzt drei Attribute. Diese sind `kind, testexpr` und `subselect`.

`kind` kann sieben Werte einnehmen, welche äquivalent zu denen in PostgreSQL sind (vgl. `SUBLINK↪:sublinktype`):

`EXISTS_SUBLINK`, `ALL_SUBLINK`, `ANY_SUBLINK`, `ROWCOMPARE_SUBLINK`, `EXPR_SUBLINK`, `ARRAY_SUBLINK` und `CTE_SUBLINK`.

In `subselect` wird die Subquery des `sublink` gespeichert.

In `testexpr` steht ein Function Call für `ALL/ANY/ROWCOMPARE`-Sublinks, welcher den entsprechenden Testausdruck darstellt.

Im Fall von `EXISTS/EXPR/ARRAY/CTE`-Sublinks steht hier ein leeres Dictionary.

5.3.18 Values

`values` entspechen dem SQL-Konstrukt *`VALUES`*.

Values werden durch `args` und `type` beschrieben.
Das Attribut `args` ist eine Liste von Listen. Jede dieser Listen steht jeweils für eine Row, jeder Eintrag in diesen Listen steht für eine Spalte. Unter `type` findet sich der Datentyp der ausgegebenen Row.

```
where
 └ expr  sublink
            ├ kind ROWCOMPARE_SUBLINK
            ├ testexpr function call
            |                ├ kind bool.and
            |                ├ args [ function call, ... ]
            |                |               ├ kind pg_operator.=
            |                |               ├ args [ colref  ,  ... ]
            |                |               |           ├ row A1
            |                |               |           ├ col rating @2
            |                |               |           └ type atom
            |                |               |                   ├ typename int4
            |                |               |                   └ typecategory N
            |                |               |
            |                |               ├ type fn
            |                |               |       ├ args [atom ,  ... ]
            |                |               |       |        ├ typename int4
            |                |               |       |        └ typecategory N
            |                |               |       |
            |                |               |       └ opres atom
            |                |               |                ├ typename bool
            |                |               |                └ typecategory B
            |                |               |
            |                |               └ kind FUNCTION_CALL_OP/BOOL
            |                ├ type fn
            |                |       ├ args [atom ,             atom]
            |                |       |        ├ typename bool    ├ typename bool
            |                |       |        └ typecategory B   └ typecategory B
            |                |       |
            |                |       └ opres atom
            |                |                ├ typename bool
            |                |                └ typecategory B
            |                └ kind FUNCTION_CALL_OP/BOOL
            |
            |
            └ subselect block
                        ├ ...
```

Abbildung 5.15: Auszug aus der Query *SELECT * FROM users WHERE (rating, stars) = (SELECT foo.rating, foo.stars FROM users as foo WHERE foo.name = stars);*

```
└ expr values
      ├ args [[ const ],         [ const ] ]
      |           ├ value 1          ├ value 2
      |           └ type atom        └ type atom
      |                  ├ typename int4    ├ typename int4
      |                  └ typecategory N   └ typecategory N
      |
      └ type row [ col ]
                  ├ name column1
                  └ type atom
                          ├ typname int4
                          └ typcategory N
```

Abbildung 5.16: Auszug aus der Query *select * from (values (1), (2)) as foo;*

5.3.19 With

Common Table Expressions werden mit dem Attribut `with` dargestellt. `with` besitzt drei Attribute: `definitions`, `recursive` und `query`.

`recursive` kann `true` oder `false` sein und gibt an, ob eine der in `definitions` gespeicherten `cte` rekursiv ist (vgl. `QUERY↪:recursive`).

In `query` steht die Query, in welcher die `definitions` gültig sind.

`definitions` ist eine Liste von `cte`. Jede `cte` steht für eine Query im With-Block (vgl. `QUERY↪:cteList`).

```
with
 |
 ├ recursive false
 ├ definitions [ cte ]
 |                ├ name test
 |               ...
 └ query block
           ├ select
           |  ├ targets [ target, ... ]
           | ...            ├ name name
           |               ...
           └ from
               └ ranges [ range ]
                            |
                            ├ row A1
                            ├ type row [ col, ... ]
                            |             ├ name name
                            |            ...
                            ├ lateral false
                            └ expr table
                                      ├ name test
                                      └ type row [ col, ... ]
                                                   ├ name name
                                                  ...
```

Abbildung 5.17: `with Beispiel`

5.3.20 CTE

Eine `cte` stellt einen einzelnen Ausdruck der *`WITH`* Clause dar (vgl. `COMMONTABLEEXPR`). Ein `cte` besitzt vier Attribute: `name`, `expr`, `recursive` und `type`.

Unter `name` steht der Name, welcher der Query in der *`WITH`* Clause gegeben wurde.

In `expr` findet sich der Ausdruck auf den sich die `cte` bezieht.

`recursive` gibt an ob diese `cte` rekursiv ist und in `type` findet sich der Datentyp des Ausdrucks der `expr`.

```
[ cte ]
   ├ name test
   ├ recursive false
   ├ type row [ col, ... ]
   |             ├ name name
   |            ...
   └ expr block
             ├ select
             |  ├ targets [ target , ... ]
             |  |             ├ name name
             | ...            └ expr colref
             |                         ├ row CTE0_A1
             |                        ...
             └ from
                └ ranges [ range ]
                              ├ row CTE0_A1
                              ├ type row [ col, ... ]
                              |             ├ name name
                              |            ...
                              ├ lateral false
                              └ expr table
                                        ├ name users
                                        └ type row [ col, ... ]
                                                      ├ name name
                                                      └ type atom
                                                              ├ typename text
                                                              └ typecategory S
```

Abbildung 5.18: `cte Beispiel`

Kapitel 6

Zusammenfassung

6.1 Mögliche Weiterentwicklungen

Problematisch erschienen die verschiedenen Darstellungen der Konstanten von PostgreSQL. Da hierzu von PostgreSQL viele verschiedene Formate verwendet werden, sind für viele Datentypen unterschiedliche Darstellungen zustande gekommen. Hierbei müssen für jeden Datentyp separat Methoden erstellt werden, um diese wieder zurück umzuwandeln. Beispielsweise wird ein *int4* anders umgewandelt als ein *char* oder *time*. In der bisherigen Implementierung sind nur einige wenige dieser Dateitypen implementiert und für die vollständige Abdeckung sollten diese noch in das Programm aufgenommen werden.

Des Weiteren gibt es noch viele weitere Strukturen, welche in Log-Files auftreten können. Hier müsste das Programm um die entsprechenden Funktionen erweitert werden, da sonst nicht für alle Querys ein reibungsloser Ablauf garantiert werden kann. Bisher sind ausschließlich die in Kapitel 3 beschriebenen Strukturen implementiert. Es bietet sich dabei an, die beiden Scource Files `primenodes.h` und `parsenodes.h` heranzuziehen und die dort aufgeführten Strukturen zu prüfen und gegebenenfalls zu ergänzen.

Die Ausgabe des Programms gibt momentan für jede Query ein seperates File aus. Um diese Daten besser verarbeiten zu können wäre es denkbar, dies umzustellen um ein geparstes Log-File in genau ein JSON-File zu extrahieren zu lassen, in welchem dann eine Liste aller Querys enthalten ist. Dies hängt allerdings vom Anwendungszweck ab.

Im Attribut `row` werden die Namen momentan vom Programm automatisch vergeben. Diese Benennung könnte bei sehr tief geschachtelten Querys zu unleserlichen Namen führen.

Bisher hat sich der Parser nur auf *`SELECT`* Statements bezogen und daher können beispielsweise keine *`INSERT INTO`* bzw. *`CREATE TABLE`* Statements ge-

parst werden. Das Programm könnte um diese Funktionen erweitert werden. Zu beachten ist dabei, dass der Parse-Tree für ***CREATE TABLE*** Statements fehlerhaft zu sein schien. Da dieser innerhalb des Attributs `:utilityStmt` abbricht den Parse-Tree auszugeben, schließt er auch die Liste im Attibut `:tableElts` nicht. Er beendet die Ausgabe des Parse-Trees mit dem Attribut `:storage` und alle bis dahin noch geöffneten Klammern werden nicht mehr geschlossen. Das dies zu Abbrüchen dieses Programms beim Parsen der Log-Files geführt hat, werden diese 'fehlerhaften' Parse-Trees momentan beim parsen übersprungen.

Das Update zu PostgreSQL 9.4 hat leider gezeigt, das sich der Parse-Tree bei Updates ändern kann. Einige Funktionen mussten bereits während der Implementierung angepasst werden. Es ist daher davon auszugehen, dass vermutlich immer wieder Programmteile erneuert werden müssen.

6.2 Fazit

Im Laufe dieser Arbeit hat sich gezeigt, dass es möglich ist, mit Hilfe der Parse-Trees der Log-Files alle notwendigen Informationen auszulesen, welche für die Erstellung eines für sich stehendes Ausgabeformates notwendig sind. Dabei wurden die gängigsten SQL-Befehle implementiert und diese auch getestet. Es ist ebenfalls möglich, die ursprünglich eingelesenen Querys wieder aus dem Ausgabeformat in semantisch äquivalente SQL-Querys umzuwandeln. Zusammenfassend kann man sagen, dass die PostgreSQL Parse-Trees effizient verwendet werden können, um Querys zu parsen.

Ich hoffe, dass sich der von mir entwickelte Parser in der Praxis bewähren und für weitere Forschungsarbeiten am Lehrstuhl für Datenbankensysteme erfolgreich eingesetzt wird.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[**1**] The PostgreSQL Global Development Group (PGDG).
PostgreSQL 9.4.1 Documentation.
http://www.postgresql.org/docs/9.4/static/index.html, 1996-2015.

[**2**] The PostgreSQL Global Development Group (PGDG).
PostgreSQL 9.4.1 Source-Code.
http://www.postgresql.org/ftp/source/v9.4.1/, 1996-2015.

[**3**] Prof. Torsten Grust
Decoding PostgreSQL's numeric format in
debug_print_parse tree output
http://db.inf.uni-tuebingen.de/attachments/
PgSQL-debug-print-parse-numeric.txt, (Torsten Grust, July 22, 2014).

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Tübingen, 31.05.2015 Unterschrift