

Universität Tübingen  
Database Systems Chair

## Diplomarbeit

# A Truly Compositional SQL Debugger

von  
Fabian Kliebhan  
Matrikelnr.: 2960436  
SS 2010

Aufgabensteller: Prof. Dr. Torsten Grust  
Abgabetermin: 27.10.2010



Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Tübingen, den 27.10.2010

## Summary

This thesis deals with the concepts of building a compositional language-level SQL Debugger, its implementation in Java and its application in practice. The main foundation that made this work possible is the *SQL2PF* project [6] where SQL queries are compositionally compiled into the *Pathfinder Algebra*. The concept and details of the implementation of the Debugger and the ideas behind it are the main topic of this thesis.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Why a SQL debugger . . . . .	5
1.2	Other work . . . . .	5
1.3	A small example . . . . .	6
1.4	Outline of the following Chapters . . . . .	7
<b>2</b>	<b>Concepts of the debugging process</b>	<b>8</b>
2.1	Building the Debugger . . . . .	8
2.2	Loop Lifting . . . . .	9
2.3	AST representation of the SQL query . . . . .	14
2.4	Compilation of the SQL AST into the Pathfinder Algebra . . . . .	17
2.5	Compilation of the <i>Pathfinder</i> plan into a new SQL query . . . . .	21
2.6	Adjustments to make debugging work . . . . .	24
2.7	Visualisation of resulting tables . . . . .	30
2.8	Implementation in Java . . . . .	32
2.8.1	The AST . . . . .	32
2.8.2	Class representation of the <i>Pathfinder Algebra</i> . . . . .	34
2.8.3	The compilation Rules . . . . .	35
2.8.4	Collecting and converting the resulting plans . . . . .	35
2.8.5	Extraction of SQL queries and execution with jdbc . . . . .	35
2.8.6	Visualization of the resulting tables using the <i>Jide</i> Framework . . . . .	37
<b>3</b>	<b>How the Debugger works</b>	<b>39</b>
3.1	An introductory example . . . . .	39
3.2	Advanced observations . . . . .	43
<b>4</b>	<b>Performance Analysis</b>	<b>46</b>
4.1	SQL to SQL compilation time . . . . .	46
4.2	Execution time of the resulting SQL code . . . . .	47
<b>5</b>	<b>Recapitulation</b>	<b>52</b>
5.1	Summary . . . . .	52
5.2	Prospect . . . . .	52
<b>A</b>	<b>Compilation Rules</b>	<b>53</b>
<b>B</b>	<b>Extended Rules for Debugging</b>	<b>62</b>
<b>C</b>	<b>TPC-H Queries</b>	<b>71</b>
	<b>References</b>	<b>91</b>

# 1 Introduction

## 1.1 Why a SQL debugger

The first question that arises is why would someone need a SQL debugger. Imagine a very common case. You got the task to write SQL queries in order to access a database and check a lot of very complex conditions that need to be fulfilled. So you're starting to write some of the queries and every time you complete a query you're testing it. Suddenly something goes wrong. E.g. rows that should be in the result set are not there, you're getting a too large result, compilation errors occur and so on. You're trying to figure out what exactly happened by looking at the query, rewriting it and testing it again but you just can't comprehend why this happened. Such bugs are very annoying and it often takes a lot of time and energy to discover what exactly went wrong. If something like this occurs again and again it becomes a really major problem and you should try to find a different way to deal with these problems than just testing and rewriting the queries. So the goal is to find and solve suchlike bugs in a very fast and easy way because of the assumption that such bugs will occur again and again.

## 1.2 Other work

Probably the first thing you would do is take a look around what options are available to deal with these problems. The following will show a few examples of available techniques for SQL debugging.

1. *The Embarcadero SQL Debugger* ([2])

This debugger offers debugging of procedures, triggers, functions and packages. It therefore helps to monitor the execution of stored procedures or other scripts. Unfortunately it lacks the possibility to debug other parts of a SQL query. So its application is basically bound to such special cases.

2. *Transact-SQL Debugger in Microsoft SQL Server 2008* ([3])

As the title already says the *Transact-SQL Debugger in Microsoft SQL Server 2008* is limited to Transact-SQL (the language of SQL Server). It also lacks the possibility to trace or inspect the invocation of a SQL query from within the Transact-SQL language. Its application is therefore bound to those special cases where you want to debug Transact-SQL code.

3. *A. Chapman and H.V. Jagadish. Why Not?* ([5])

The debugging technique of by A. Chapman and H.V. Jagadish explains the absence of expected rows. However, it works at the level of algebraic primitives and is therefore disconnected from the SQL query syntax.

```
SELECT r_regionkey
FROM   region
WHERE  r_name = 'EUROPE';
```

Figure 1: Query  $Q1$

We can conclude that all the previously described techniques don't really offer an appropriate solution to the described problem. Imagine again what you would normally do to find a bug. The main thing you're probably interested in are partial result of the query. So you're rewriting your query to get results of subparts in order to track the problem down to a minimal core. Or you're rewriting the query to understand more exactly what this subpart actually did and how it affected the rest of the query. Either way it seems to be a very useful approach to narrow down the problem to subparts of the query. The problem with this approach is that it is a very exhausting and time consuming job and therefore no good solution for recurring problems. The ideal would be an automated process that rewrites parts of the query itself while the user just marks different subexpression and this process generates a corresponding SQL query. The Compositional Debugger offers exactly this solution. You can mark every desired subexpression (literals, predicates or even subquery blocks) directly on the language-level. The Debugger then creates corresponding SQL queries and executes them on the same database you're already using and you can take a look at these partial results. This makes it very easy to discover what went wrong. Also no further understanding of the underlying relational algebra is necessary in order to use the Debugger and no additional debugging middleware is required.

### 1.3 A small example

Take a look at query  $Q1$  in Figure 1. The bug in this query is a very simple misspelling. Instead of 'EUROPE', 'Europe' should be used in the WHERE block. For the query writer the only hint that a bug is present is that the resulting table is empty. With the help of the Compositional Debugger one can mark the suspect subexpression where the bug is assumed (here e.g. the predicate:  $r\_name = 'EUROPE'$ ) and take a look at the resulting tabular representation after this subexpression is evaluated. One can observe that this predicate is always evaluated to *false* because 'EUROPE' doesn't equal any  $r\_name$  row. Therefore one can conclude that the spelling was wrong. A detailed investigation of this query is given in Chapter 3.

As we have seen so far this Debugger offers a great solution to our problem. It is very easy to understand and use and it saves a lot of time compared to the effort that occurs by rewriting many queries manually.

Another application field of the Debugger is teaching. It can help to understand

the compositional execution of a SQL query and can therefore be used for teaching purposes in database lectures.

#### **1.4 Outline of the following Chapters**

Chapter 2 deals with the underlying concepts of the Debugger. It shows how such a Debugger can be built and how it internally works. If you're interested in the application of the Debugger and how you can use it to observe your own query you should read Chapter 3. Chapter 4 gives a performance analysis of the Debugger and can be read independently of the other Chapters. Chapter 5 summarizes the previous Chapters and gives a prospect of further work.

```

SELECT  n_name
FROM    nation
WHERE   EXISTS
        (SELECT r_name
         FROM  region
         WHERE {r_regionkey = n_regionkey
              AND r_name = 'EUROPE'});

```

Figure 2: Example query  $Q2$ . The braces mark the interesting subexpression.

nation		region	
n_name	n_regionkey	r_regionkey	r_name
Argentina	1	0	Africa
Germany	3	1	America
Japan	2	2	Asia
		3	Europe

Figure 3: The tables used in the example: *nation* and *region*

## 2 Concepts of the debugging process

### 2.1 Building the Debugger

The application of the Debugger is very clear. You mark parts of the queries and take a look at the resulting output of the marked subexpression. To implement this, the obvious idea is to extract the marked subexpressions from the base SQL query and make adjustments in order to convert it into a new, syntactical correct SQL query. Consider the buggy query  $Q2$  in Figure 2 (the bug is the same as before. 'EUROPE' is used instead of 'Europe' which is an entry in the *region* table). The resulting rows of query  $Q2$  should be the names (*n\_name*) of every country in the *nation* table belonging to the region 'Europe'. The tables *region* and *nation* are shown in Figure 3. In order to trace down the bug, one might want to look at the tabular results of the evaluation of the predicates in the WHERE clause of the subquery (this part is highlighted with braces in Figure 2). The WHERE clause of the subquery references the *n\_regionkey* column of the *nation* table. The subquery therefore depends on the outer SFW (SELECT-FROM-WHERE) block where the *nation* table is accessed. To be exact, it depends on the implicit introduced *row variable*, that represents this occurrence of the *nation* table (every table or subquery referenced by a FROM clause introduces implicitly a *row variable*. The corresponding FROM clause is called *binding site* of that *row variable*). Therefore the subexpression can't be evaluated independent of its surrounding and a restriction on this subexpression is insufficient. So, correct queries for subexpressions can't be generated by simple extraction. An approach that wants to create a valid query of such a subexpression must always consider the surroundings, namely

Operator	Semantics
$\pi_{a_1:b_1, \dots, a_n:b_n}$	project onto columns $b_i$ (and rename into $a_i$ )
$@_{a:v}$	attach a column $a$ with the constant value $v$
$- \times -$	Cartesian product
$\lambda_{a:f\langle b_1, \dots, b_n \rangle}$	apply $n$ -ary scalar function $f$ and attach the result in $a$
$\sigma_p$	eliminate rows that don't satisfy predicate $p$
$- \bowtie_{a=b} -$	equi-join on columns $a$ and $b$
$- \cup -, - \setminus -$	disjoint union, difference
$\delta$	eliminate duplicate rows
$AGG_{a:\langle b \rangle/c}$	group the rows by $c$ and attach aggregate of $b$ in $a$
$\#_a$	attach unique row identifier in $a$
$\vec{\#}_{a:\langle b_1, \dots, b_n \rangle}$	attach row number in $b_1, \dots, b_n$ order in $a$
$\varrho_{a:\langle b_1, \dots, b_n \rangle}$	attach row rank in $b_1, \dots, b_n$ order in $a$
$TABLE_{b_1, \dots, b_n}(t)$	access rows in table $t$ and name the columns $b_1, \dots, b_n$
$LIT\_TBL_{a:[a_1, \dots, a_n], b:[b_1, \dots, b_n], \dots}$	create table with rows $a, b, \dots$ and corresponding entries $a_1, \dots, a_n, b_1, \dots, b_n, \dots$
$SERIALIZE_{a,b,\langle c_1, \dots, c_n \rangle}$	create the result table with columns $c_1, \dots, c_n$ . $a$ ( <i>iteration</i> ) and $b$ ( <i>position</i> ) are responsible for the concrete order.

Table 1: Excerpt of the *Pathfinder Algebra*  
(AGG  $\in \{COUNT, SUM, MAX, MIN, AVG, ANY\}$ ,  $f \in \{+, -, =, <, \dots\}$ )

the *free row variables* (*row variables*, that are bound in outer SFW clauses). This is necessary because of the structure of SQL evaluation, where every expression is evaluated in dependance of these *row variables*.

A solution is a compositional approach that is fully aware of all *row variables*.

## 2.2 Loop Lifting

*Pathfinder* can help building such a solution. *Pathfinder* is a query compiler that turns XQuery expressions compositionally into queries of its own Table Algebra (an excerpt with important operators of the *Pathfinder Algebra* is shown in Table 1). The compilation uses an approach called *Loop Lifting* to compile the XQuery expressions. This technique can be adapted in order to use SQL queries instead of the XQuery expressions and compile SQL subexpressions fully aware of the depending *row variables* compositionally into the *Pathfinder Algebra*. The generated algebraic code, in turn, can be used to produce tabular expressions that correspond to subresults of the SQL evaluation process at an arbitrary point. As we have seen in the Section 2.1, in order to emit the correct tabular representation for subexpressions, the relationship to outer SFW blocks (also called *scopes*), if present, must be represented correctly.

This is done by compiling inner query blocks in dependence of the *row variables* of the outer query blocks. Recall again the highlighted subexpression in Figure 2. Algebraic code that corresponds to this subexpression must at first represent the relationship of the *region* table to the outer SFW clause correctly. Only after this is done, the algebraic code can be extended to deal with the predicates in the WHERE clause of the subquery correctly. To achieve this correct representation, the *region* table is compiled in dependence of the *nation* table *row variable*.

Figure 4 shows the concrete algebraic code (the used algebraic operators are explained in Table 1). As you can see, it starts with a single-column, single-row table with entry 1. At first a cross product is made with the *nation* table. At this point, the algebraic code represents the top-level FROM clause. After that a cross product is made of the unique row identifier column *ik* of the current tabular representation (created by the # operator) and the *region* table. This lifts this row identifier. After that an Equijoin is made in order to adjust the *nation* table columns to it's lifted identifier. In general, the algebraic code that is used to accomplish the lifting technique is given by the following function.

$$LIFT(o, i) \equiv \#_{ik}(\pi_{ok:ik}(o) \times i)$$

*ik* is a special column that stands for *inner key* and will be used as a key of the current scope during the compilation process. Another special column is the *outer key* column (*ok*) that will be used as reference to the outer scope.

Take again a look at Figure 4. Like already mentioned the algebraic evaluation begins with a single-row, single-column table with entry 1 (called  $e_0$  in Figure 4). This table is created by the algebraic operator  $LIT\_TBL_{ik:[1]}$ . The algebraic code representing the top-level FROM clause can be derived using the  $LIFT$  function on  $LIT\_TBL_{ik:[1]}$  and the *nation* table (to access the *nation* table the algebraic  $TABLE$  operator is used).

$$\begin{aligned} e_1 &\equiv LIFT(LIT\_TBL_{ik:[1]}, TABLE_{i_1, i_2}(nation)) \\ &\equiv \#_{ik}(\pi_{ok:ik}(LIT\_TBL_{ik:[1]}) \times TABLE_{i_1, i_2}(nation)) \end{aligned}$$

To derivate algebraic code corresponding to the FROM clause of the subquery the  $LIFT$  function can be used again. This time on  $e_1$  and the *region* table.

$$\begin{aligned} e_2 &\equiv LIFT(e_1, TABLE_{i_3, i_4}(region)) \\ &\equiv LIFT(LIFT(LIT\_TBL_{ik:[1]}, TABLE_{i_1, i_2}(nation)), TABLE_{i_3, i_4}(region)) \\ &\equiv \#_{ik}(\pi_{ok:ik}(\#_{ik}(\pi_{ok:ik}(LIT\_TBL_{ik:[1]}) \times TABLE_{i_1, i_2}(nation))) \\ &\quad \times TABLE_{i_3, i_4}(region)) \end{aligned}$$

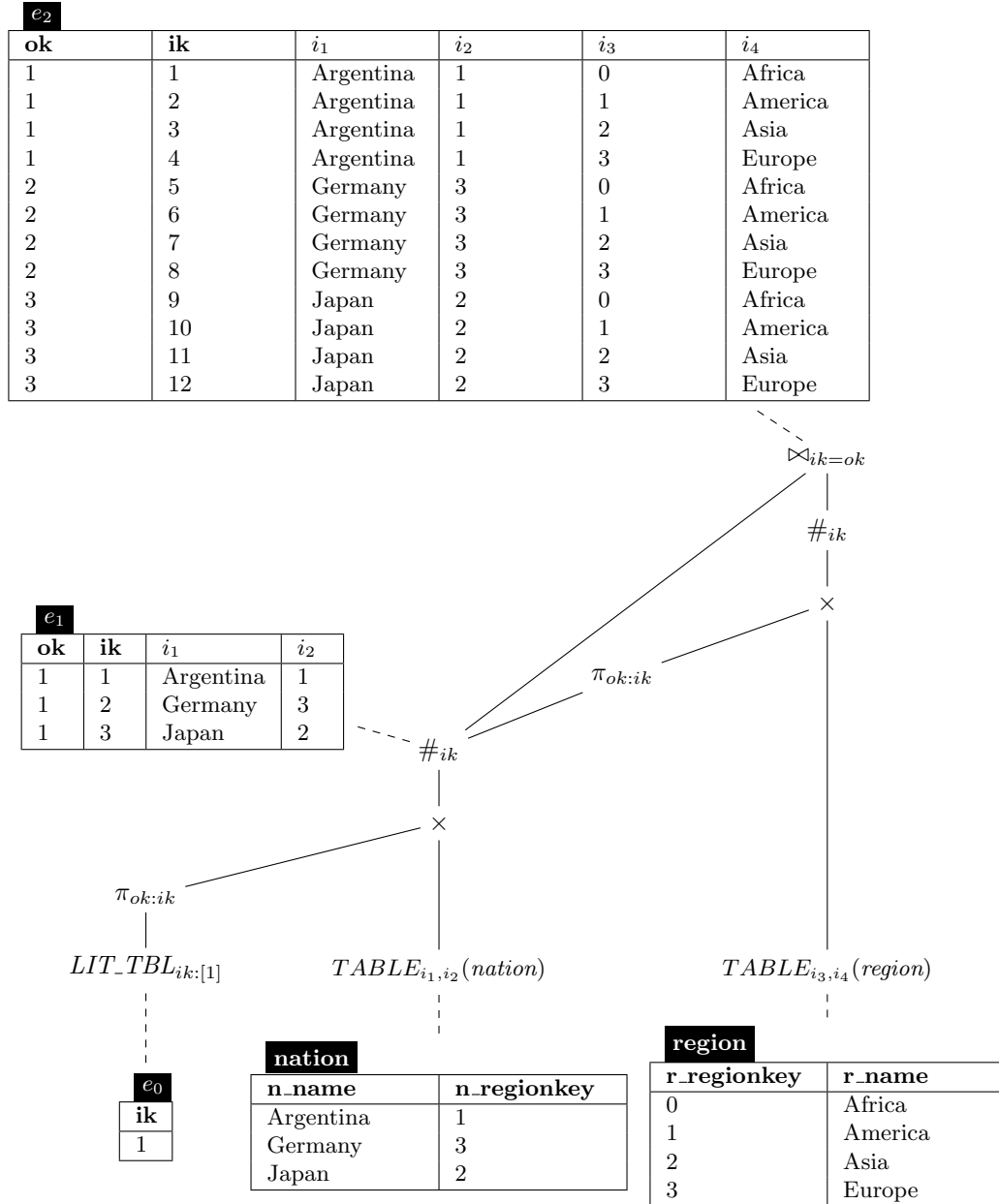


Figure 4: Algebraic Code that lifts up the *nation* table.

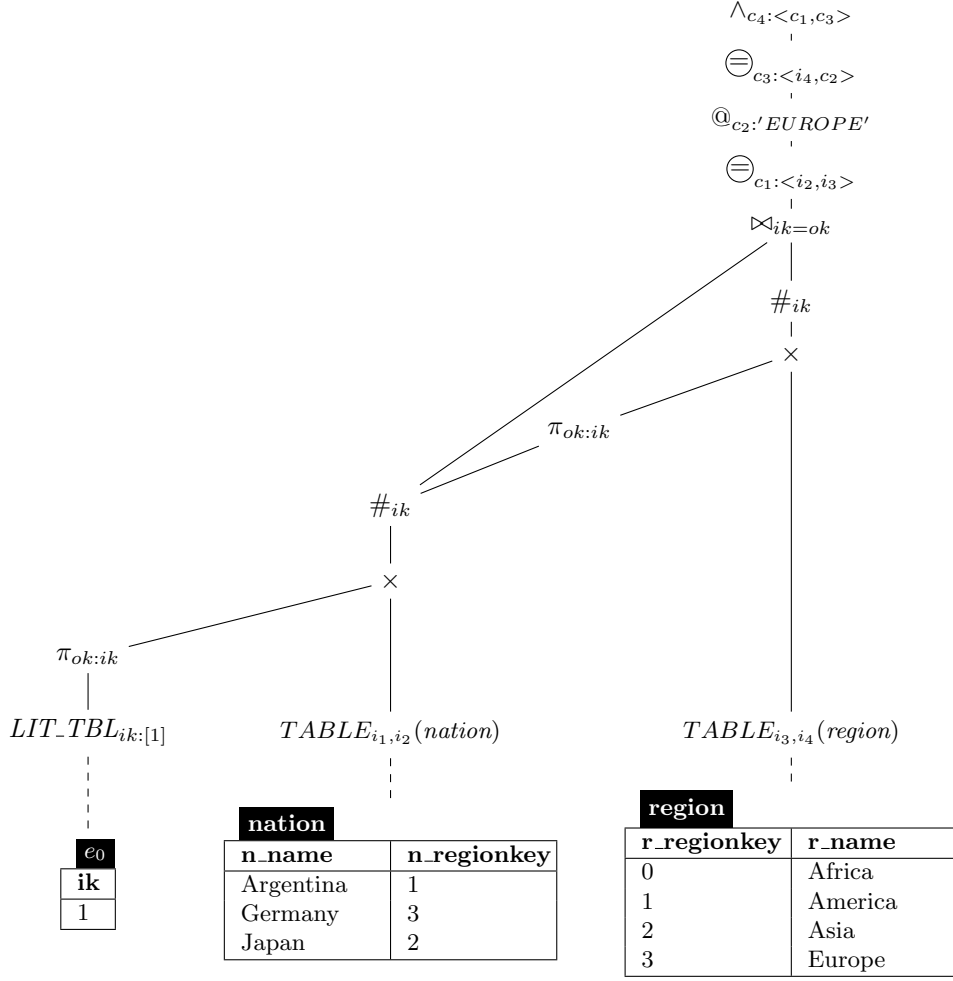


Figure 5: Algebraic code of the *Pathfinder Algebra* corresponding to the marked subexpression of query  $Q_2$  in Figure 2

This is the algebraic code shown in Figure 4 except the adjustments of the *nation* table rows. Now that the relationship to the outer SFW clause is represented correctly the algebraic plan can be extended to represent the marked subexpression of Figure 2. Figure 5 shows that extended algebraic code. The corresponding tabular result that is generated by evaluating this algebraic code is given in Figure 6. As we have seen this lifting techniques makes it possible to evaluate subexpressions in relation to their outer scopes and can therefore be used in order to create compilation Rules that can compile an arbitrary SQL query into algebraic code of the *Pathfinder Algebra*.

So far it has been said a few times that this algebraic code can be evaluated. This

ok	ik	$i_1$	$i_2$	$i_3$	$i_4$	$c_1(i_2 = i_3)$	$c_2$	$c_3(i_4 = c_2)$	$c_4(c_2 \wedge c_3)$
1	1	Argentina	1	0	Africa	false	EUROPE	false	false
1	2	Argentina	1	1	America	true	EUROPE	false	false
1	3	Argentina	1	2	Asia	false	EUROPE	false	false
1	4	Argentina	1	3	Europe	false	EUROPE	false	false
2	5	Germany	3	0	Africa	false	EUROPE	false	false
2	6	Germany	3	1	America	false	EUROPE	false	false
2	7	Germany	3	2	Asia	false	EUROPE	false	false
2	8	Germany	3	3	Europe	true	EUROPE	false	false
3	9	Japan	2	0	Africa	false	EUROPE	false	false
3	10	Japan	2	1	America	false	EUROPE	false	false
3	11	Japan	2	2	Asia	true	EUROPE	false	false
3	12	Japan	2	3	Europe	false	EUROPE	false	false

Figure 6: Resulting table if algebraic code in Figure 5 is evaluated

becomes possible because *Pathfinder* brings it's own SQL generator. So, in order to evaluate the algebraic plan, it gets translated back into a SQL query, that can be executed directly on your database host. No further middleware is required.

Figure 7 shows the complete compilation process.

The following three sections (2.3,2.4,2.5) will explain the compilation process from SQL to SQL. First an overview over the AST representation of the SQL query is given. Then the compilation process that compiles the SQL AST into the *Pathfinder Algebra* plan is covered and the compilation back to SQL is explained. Because so far nothing is won (we compiled SQL to SQL) Section 2.6 explains the use case of the Compositional Debugger. Also, the changes, that need to be made to the compilation process in order to make debugging work, are explained.

Section 2.7 shows how to present resulting tables of different observations in relation to each other and Section 2.8 deals with the concrete implementation of the so far explained concepts in Java.

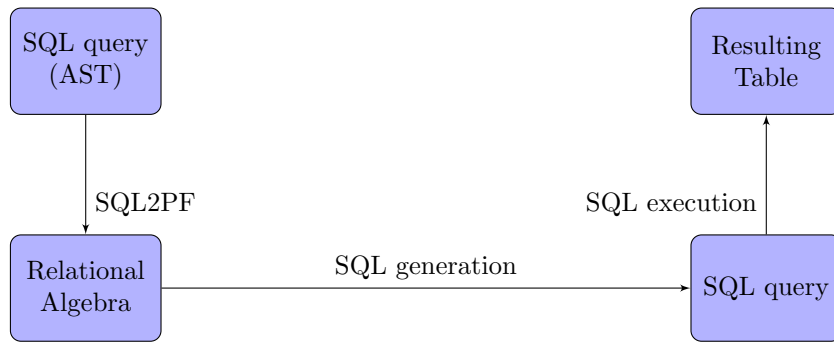


Figure 7: **Compilation process.** The SQL query is present as an AST (Abstract Syntax Tree). This AST is compiled by Compilation Rules. These Rules create a *Pathfinder Algebra* plan. The process is titled **SQL2PF**. This plan can be optimized and compiled back into a SQL query. This is done by the *Pathfinder* tools *pfopt* and *pfsql*. We call this process **SQL generation**. Finally in the **SQL execution** step, this resulting SQL query can be executed by the database host in order to get the resulting tables.

### 2.3 AST representation of the SQL query

The SQL AST (Abstract Syntax Tree) representation contains components that encapsulate the different expressions that appear in a SQL query (like e.g. literals, predicates or subqueries). Also such an AST representation of a SQL query is directly used by the compilation into the *Pathfinder Algebra*. A parser, that parses the SQL query into the corresponding AST is already given (see [6]). As an example the AST representation of SQL query *Q1* from Figure 1 is shown in Figure 8.

The following AST components exist.

- **The SFW component**

The *SFW* component represents a SFW clause in a SQL query. It encapsulates different substructures which are a *FromBlock* component, a *WhereBlock* component, a *SelectBlock* component, *GroupBy* and *Having* components and a *OrderBy* component (not all must be given).

- **The FromBlock component**

The *FromBlock* component, in turn, encapsulates AST components representing the elements in the FROM clause (strictly speaking the referenced tables or subqueries). These components can be of the type *Table* or *SFW*.

- **The WhereBlock component**

This component encapsulates another AST component. This AST component can be of different types depending of the concrete expression. E.g. if the WHERE clause starts with an EXISTS expression, the encapsulated AST component is an *Exists* component.

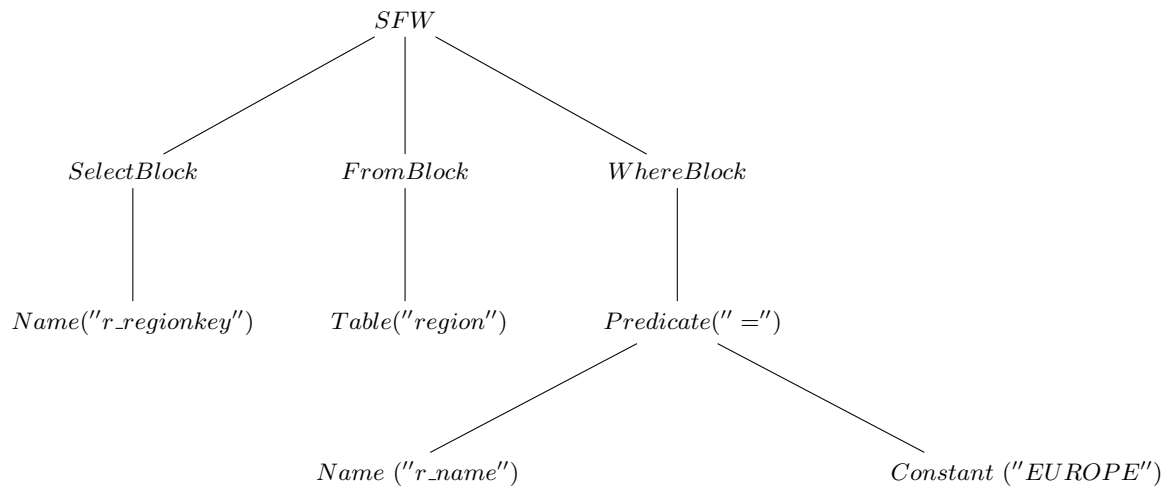


Figure 8: SQL AST class representation of query  $Q1$

- **The SelectBlock component**

The *SelectBlock* component encapsulates those components that represent the columns referenced in the SELECT block.

- **GroupBy and Having components**

*GroupBy* component represents the referenced GroupBy columns. A *Having* component encapsulates exactly one AST component. The concrete type depends on the expression.

- **The OrderBy component**

A *OrderBy* component encapsulates *Name* components representing the referenced columns of the OrderBy clause.

- **The Predicate component**

A *Predicate* component encapsulates the underlying AST components. The *Predicate* instance representing the expression  $r\_name = 'EUROPE'$  e.g., encapsulates a *Name* (for  $r\_name$ ) and *Constant* (for 'EUROPE') component.

- **The Table component**

The *Table* component represents a SQL table.

- **The Constant component**

This component represents number, string or date constants.

- **The Name component**

A *Name* component represents the corresponding column referenced inside a query. If the same column is referenced twice inside a query, two different *Name* components with the same column name are used.

- **All, Any, Case, Exists and In components**

These components represent the corresponding operators. They encapsulate the AST components representing the referenced subexpression.

$ \begin{aligned} m_1 &\equiv i_{New()} \\ &\vdots \\ m_n &\equiv i_{New()} \\ q' &\equiv \#_{ik}(@_{s:1}(\tau_{ok:ik}(q) \times TABLE_{m_1, \dots, m_n}(e))) \\ \Gamma' &\equiv \{ e.col_1 \rightarrow m_1, \dots, e.col_n \rightarrow m_n \} \end{aligned} $ <hr style="border: 0.5px solid black;"/> $(\Gamma, q); \{ \} \vdash e(col_1, \dots, col_n) \Rightarrow (\Gamma', q')$
--

**Equation 1:** Table Rule.

## 2.4 Compilation of the SQL AST into the Pathfinder Algebra

The compilation process is based on different compilation Rules (also described in [7],[8],[6]) corresponding to the different parts of the SQL AST. These Rules describe how a SQL query (represented as AST) can be transformed into a sequence of algebraic operators (see Table 1) in compositional manner. The Rules call other Rules in order to deal with subexpressions of the current AST block. This is done generically, which means that depending on the type of the AST block, the appropriate Rule is called. In order to have a defined in- and output we need data structures to carry the algebraic plans and meta data. Two data-structures are used.

- The *Operator* structure ( $q$ ) is used to collect all needed algebraic operators and therefore carries the algebraic plan.
- The *Environment* ( $\Gamma$ ) stores references to the currently visible columns and keeps the relationship to the real table columns names. It therefore carries the meta data.

The two special columns  $ik$  and  $ok$  are used to keep the relationship of different scopes. The  $ik$  column is always a key of the current scope. The  $ok$  always refers to the outer scope of the current scope. To connect different scopes, joins on these keys are necessary. The special column  $s$  is used to introduce a order on the rows. This becomes necessary e.g. if a *OrderBy* statement is present.

There are two types of Rules. One type is called *Table Inference Rule* and returns references to more than one column (represented by the *Environment* data structure) and the other is called *Column Inference Rule* and returns only a single column reference. Furthermore both Rules return the *Operator* data structure. The input of all Rules is always an *Environment* and an *Operator*. E.g. the *Constant Rule*, that is called when compiling a *Constant* instance of the AST, always returns a single column reference (corresponding to the column containing the at-

tached constant). The Table Rule, on the other hand, returns an *Environment*, because references to all columns of the accessed table are needed.

The input/output behavior of those Rules returning a table reference is denoted as follows:

$$(\Gamma, q), Tdqd \vdash SQLAST \Rightarrow (\Gamma', q')$$

This notation means that the Rule gets an *Environment* and *Operator*  $(\Gamma, q)$  as input and returns, again an *Environment* and *Operator*  $(\Gamma', q')$ . The additional input  $Tdqd$  is used in queries with aggregations where the relationship to the un-aggregated state must be kept (we ignore this input for now). Another input, emphasized by the  $\vdash$ , is the AST node ( $SQLAST$ ) that is currently compiled. The input/output behavior of the Rules returning a column reference is denoted a little different:

$$(\Gamma, q), Tdqd \vdash SQLAST \triangleright (q', col)$$

As you can see the input is handled the same way as before. The output on the other hand is now a single column reference ( $col$ ) and again an *Operator* ( $q'$ ). This denotation of the input/output behavior is shown on the bottom of every Rule. The concrete sequence of composed algebraic operators, created by the Rules, are listed above this denotation. If another Rules is called (generically) inside of a Rule the same denotation is used.

Equation 1 shows the TABLE Rule as an example to point out how the Rules work. As you can see the Table Rule input is  $(\Gamma, q)$  and the current AST component (*Table e* with columns  $col_1, \dots, col_n$ ). The output is  $\Gamma'$  and  $q'$ . This output is created by the operations listed above the input/output denotation. The *TABLE* operator accesses table  $e$  and renames its columns to  $m_1, \dots, m_n$  (These unique column names are created using  $i_{New()}$ ). Then a cross product of the current row identifier column and the accessed table is made. The old row identifier column ( $ik$ ) becomes thereby the new *outer key* column ( $ok$ ). This corresponds exactly to the *Loop Lifting* technique described in Section 2.2. After that, the  $s$  column with row values  $1$  (needed later by the *Serialize* operator) is attached and a new row identifier ( $ik$ ) is created. The algebraic code of these operations is stored in  $q'$ . References to the columns  $m_1, \dots, m_n$  and the corresponding real column names  $col_1, \dots, col_n$  are stored in  $\Gamma'$ .

A complete list of all Rules is given in Appendix A. One can see that the compilation is done compositionally. The concrete compilation of a specific SQL query (represented by a SQL AST) starts with the *Serialize* Rule. This Rule first emits algebraic code for the single-column single-row table with entry  $1$  (the initial tabular expression) and then calls the *SFW* Rule for the *SFW* AST node representing the whole query. The *SFW* Rule directly calls the *FROM* Rule which appends a

$\pi$  operator and, in turn, calls either the Table Rule (if the first referenced FROM item is a table) or the SFW Rule (if the first referenced FROM item is a subquery). This process continues compositionally until the whole AST is compiled into algebraic code. At last the compilation returns to the Serialize Rule and the algebraic code is completed by adding the *Serialize* operator (this operator is responsible for outputting the tabular result). Figure 9 shows the algebraic code that is created by compiling query *Q1* of Figure 1 using these Rules. The corresponding Rules, responsible for the different segments of algebraic code (stored in the *Operator* structure) are indicated. The meta content of the *Environment* structure is not shown here. The Serialize Rule creates the *LIT\_TBL* operator for the starting single-row, single-column table and starts the SFW Rule. The SFW Rule directly calls the FROM Rule that calls, in turn, the Table Rule. The  $\times$  operator is added to join the *region* table onto the current tabular expression (this corresponds to the *Loop Lifting* concept shown in Section 2.2). The Table Rule also produces a  $\#$  and  $@$  operator to attach the two necessary columns *ik* and *s* (these columns need to be present at the end of every Rule because the *ik* column could be needed by outer scopes and the *s* column is needed by the *Serialize* operator). Back in the FROM Rule, different operators are appended to collect all columns (becomes important if the FROM clause contains more than one item). The SFW Rule, in turn, unites the columns of the current scope with those from the outer scope (important if the current SFW block is not on the top-level). Now that all columns are collected, the WHERE Rule is started. It calls the Predicate Rule. Because a constant ('EUROPE') is present, the Constant Rule is started. The  $@$  operator is added to attach that constant. Back in the Predicate Rule, code for the equal-predicate is created. After that, the WHERE Rule applies the  $\sigma$  operator in order to discard those rows that were evaluated to false by the equal-predicate. Then the SELECT Rule is called to produce the  $\pi$  operator for the columns in the SELECT clause ( $i_1$ ). At last the Serialize Rule creates the *SERIALIZE* operator that is used to produce the output.

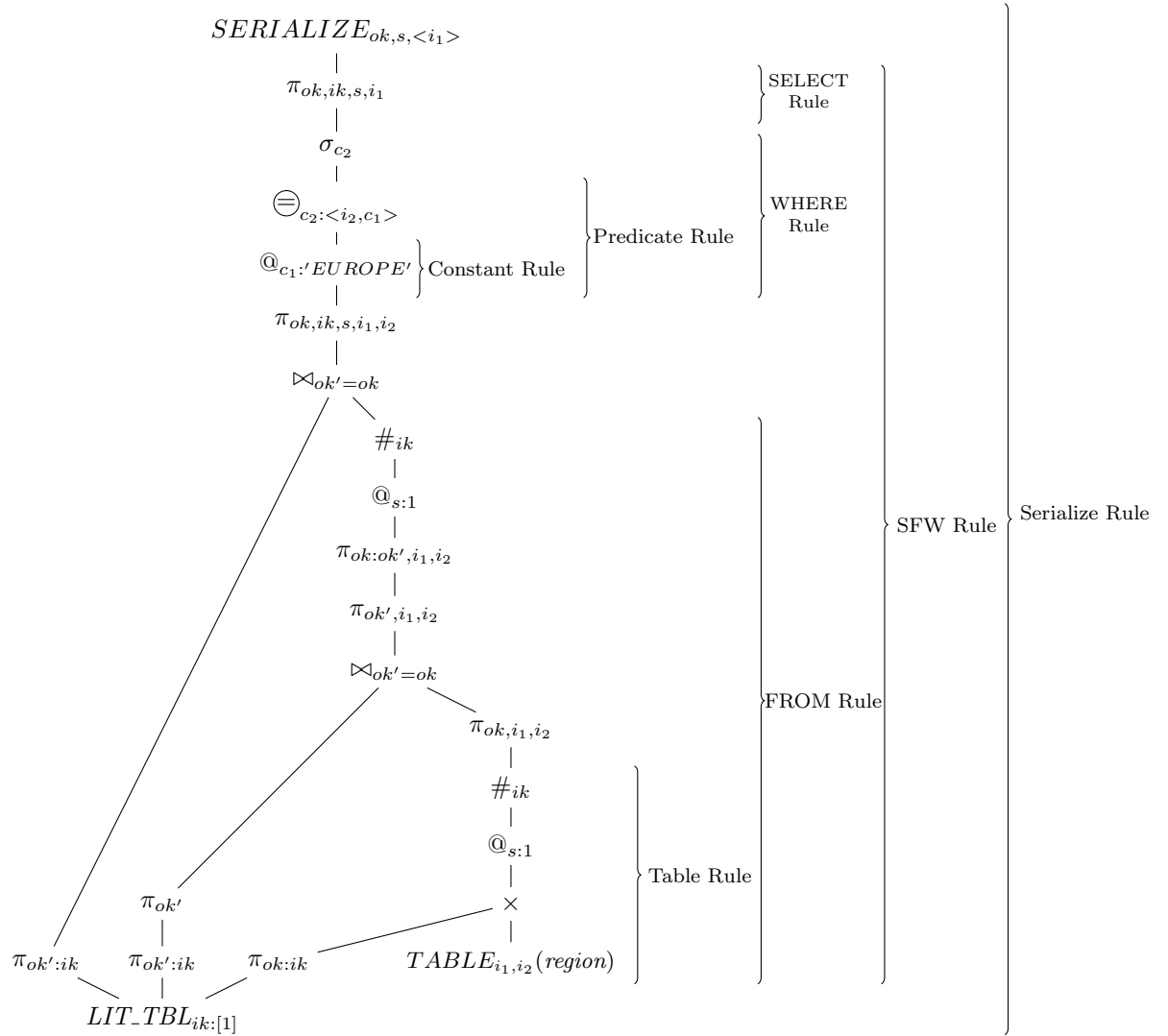


Figure 9: Algebraic code of the *Pathfinder Algebra* corresponding to query  $Q1$  in Figure 2 created by the compilation Rules in Appendix A.

## 2.5 Compilation of the *Pathfinder* plan into a new SQL query

After the algebraic plan is complete, it can be compiled into a new SQL query, that can be executed on the database host. There are two tools that are used to accomplish this compilation.

- *pfopt* is used to generate an optimized, semantically equal version of an algebraic plan.
- *pfsql* compiles algebraic plans into SQL code.

We use these tools on the plan created by the compilation Rules. At first *pfopt* is applied onto this plan. The output is, again, a valid algebra plan, that equals semantically the resulting plan of the compilation Rules. Onto this newly created plan we apply *pfsql* to create SQL code that can be executed on the database host. Figure 11 shows the optimized version of the *Pathfinder Algebra* plan given in Figure 10 after *pfopt* is applied. These plans correspond (again) to query *Q1* from Figure 1. We can see that this optimization has a major impact on the plansize and is therefore a crucial factor to the overall performance.

Figure 12 shows the resulting SQL query when applying *pfsql* onto the optimized plan of query *Q1*. As you can see the output basically equals query *Q1*.

So far nothing is won. We compiled SQL into the *Pathfinder Algebra* and back into SQL. One use case of this compositional SQL compiler is the Debugger. To make debugging work a few adjustments need to be made. These adjustments are topic of the following section.

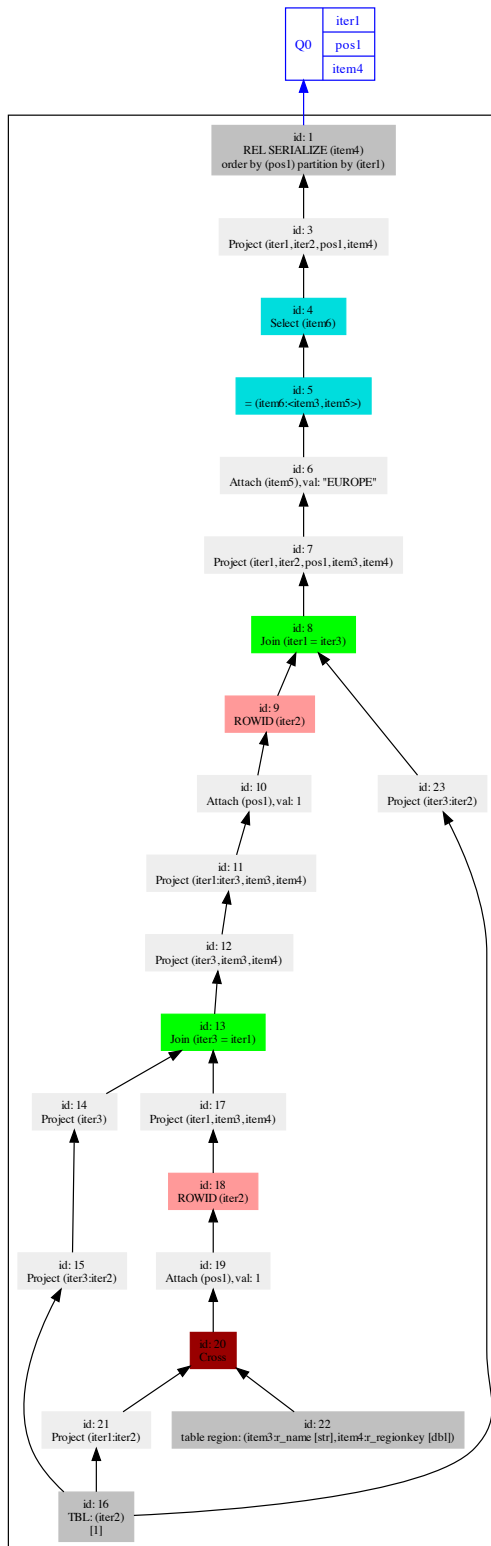


Figure 10: Optimized *Pathfinder Algebra* Plan of query Q1

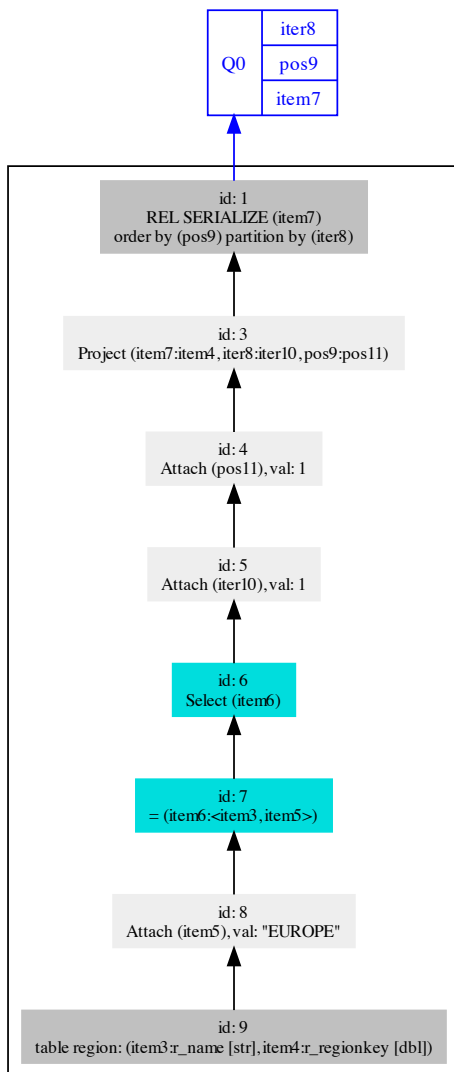


Figure 11: Optimized *Pathfinder Algebra* Plan of query *Q1*

```

SELECT  a0000.r_regionkey AS item7_int
FROM    region AS a0000
WHERE   a0000.r_name = 'EUROPE';

```

Figure 12: Resulting SQL query when compiling query *Q1*

```

SELECT  r_regionkey
FROM    region
WHERE   r_name = 'EUROPE';

```

Figure 13: Marking of the equal-predicate and the 'EUROPE' constant of query  $Q1$

## 2.6 Adjustments to make debugging work

The following will show adjustments to all parts of the compilation process. Unlike before the starting point of the debugging process is a query with different markings (the normal compilation process does not consider markings).

First, the markings of the underlying SQL query become part of the AST. They are represented as *Debug* nodes and encapsulate the marked subexpression. This way, the compilation Rules can handle the *Debug* node like every other AST node and collect the already constructed plan whenever a *Debug* node appears. E.g. Figure 14 shows the resulting SQL AST if the equal-predicate and the 'EUROPE' constant in query  $Q1$  are marked (these markings are given in Figure 13).

To handle the *Debug* node in the *AST2PF* compilation two new Rules, called *DebugTableRule* and *DebugColumnRule*, are introduced:

$(\Gamma, q), Tdqd, Tq \vdash e \Rightarrow (\Gamma', q'), Tq'$ $Tq'' \equiv Tq' + \{(\Gamma', q')\}$ <hr style="border: 0.5px solid black;"/> $(\Gamma, q), Tdqd, Tq \vdash \text{DEBUG } e \Rightarrow (\Gamma', q'), Tq''$
---

**Equation 2:** Debug Table Rule.

$(\Gamma, q), Tdqd, Tq \vdash e \triangleright (q', c), Tq'$ $Tq'' \equiv Tq' + \{(\{col \rightarrow c\}, q')\}$ <hr style="border: 0.5px solid black;"/> $(\Gamma, q), Tdqd, Tq \vdash \text{DEBUG } e \triangleright (q', c), Tq''$
---

**Equation 3:** Debug Column Rule.

The *DebugTableRule* Rule is used if the Rule for the marked subexpression is a *Table Inference Rule* and the *DebugTableRule* Rule is used if is a *Column Inference Rule*.

Both Rules first call this Rule for the marked subexpression. This Rule creates the corresponding algebraic plan for this subexpression. In the Debug Table Rule the produced plan ( $q$ ) and the corresponding *Environment* ( $\Gamma$ ) are enqueued into

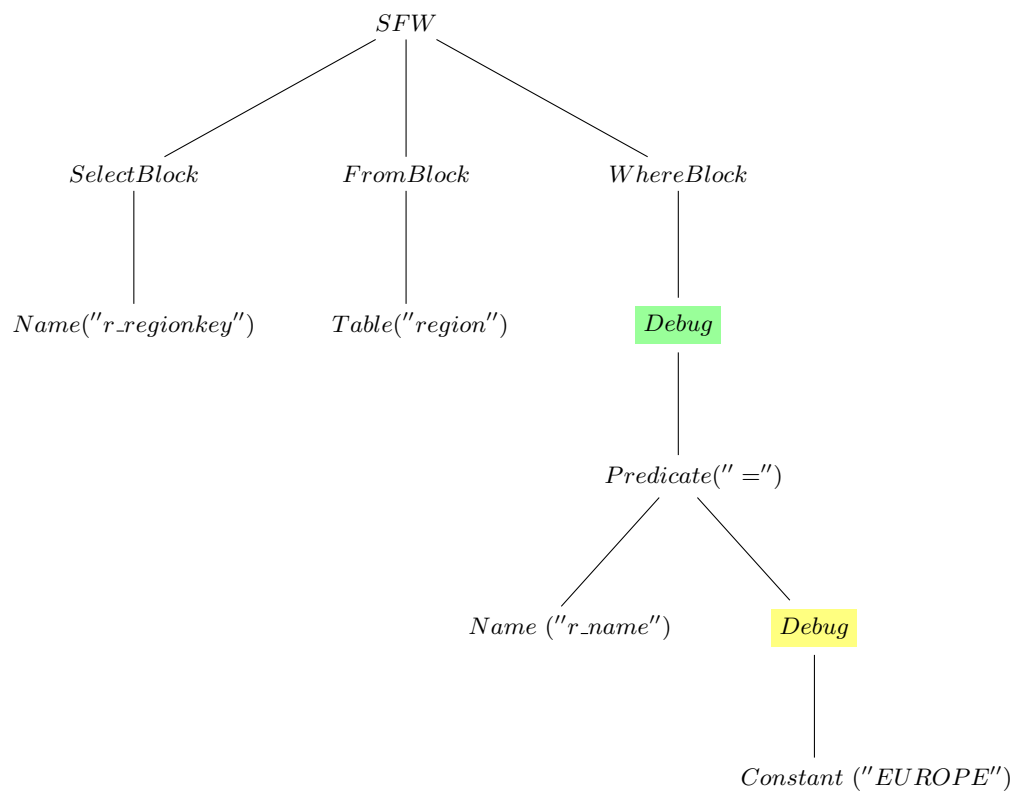


Figure 14: SQL AST class representation of query  $Q1$  with the equal-predicate and the 'EUROPE' constant marked for debugging

a data structure ( $Tq$ ) that is used to collect all debugging plans. The *Debug-ColumnRule* does nearly the same. The only difference is that it creates a new *Environment* containing the referenced column produced by the *Column Inference Rule* of the marked subexpression.

The *Serialize Rule* is modified in order to initialize the new data structure before the other Rules are started. It also adds the *Serialize* operator onto every collected item of the  $Tq$  data structure at the end of the compilation. The following shows this adapted version of the *Serialize Rule*:

$$\begin{array}{l}
(\{\}, LIT\_TBL_{ik:[1]}, \{\}, \{\}, \vdash e \Rightarrow (\Gamma', q'), Tq' \\
Tq' \equiv \{(\Gamma_{D_1}, q_{D_1}), \dots, (\Gamma_{D_n}, q_{D_n})\} \\
\forall i \in (1, \dots, m) \left\{ q'_{D_i} \equiv SERIALIZE_{ok, ik', <\Gamma_{D_i}>}(Q_{ik':<s, ik>} q_{D_i}) \right. \\
\hline
\vdash e \Rightarrow^+ (\Gamma_{D_1}, q'_{D_1}), \dots, (\Gamma_{D_m}, q'_{D_m})
\end{array}$$

**Equation 4:** Serialize Rule.

As you can see a empty  $Tq$  data structure is forwarded to the Rule called by the *Serialize Rule*. The output of that Rule contains all collected Debug plans ( $Tq'$ ). These plans are serialized and returned. In order to forward and collect the  $Tq$  data structure, the input/output behavior of all Rules and the way of calling new Rules need to be modified. All the adapted Rules are shown in Appendix B. Another thing needed, in order to put different observations in correct relation to each other, is a strict ordering of the rows. To accomplish that, the *Table Rule* and *FROM Rule* are extended. The following shows the extended *Table Rule*:

$$\begin{array}{l}
m_1 \equiv i_{New()} \\
\vdots \\
m_n \equiv i_{New()} \\
q' \equiv \vec{\#}_{ik:<ok, Keys(e)>} (@_{s:1}(\pi_{ok:ik}(q) \times TABLE_{m_1, \dots, m_n}(e))) \\
\Gamma' \equiv \{ e.col_1 \rightarrow m_1, \dots, e.col_n \rightarrow m_n \} \\
\hline
(\Gamma, q), Tdq, Tq \vdash e(col_1, \dots, col_n) \Rightarrow (\Gamma', q'), Tq
\end{array}$$

**Equation 5:** Table Rule.

The main difference is, instead of a random order, that was created before with the  $\#$  operator, now a strict ordering is done by using the  $\vec{\#}$  operator on the outer key and all key columns of the accessed table (if no key column is present, all columns are used). To retain this strict ordering, the *FROM Rule* is also adapted.

$$\begin{array}{l}
\Gamma'_0 \equiv \{\} \\
q'_0 \equiv \pi_{ok':ik,ik}(q) \\
Tq_0 \equiv Tq \\
\forall i \in (1, \dots, n) : \begin{cases}
(\Gamma, q), Tdqd, Tq_{i-1} \vdash e_i \Rightarrow (\Gamma_i, q_i), Tq_i \\
\Gamma'_i \equiv \Gamma'_{i-1} + \Gamma_i \\
q''_i \equiv \vec{\#}_{ik:\langle ik', ik'' \rangle} (\pi_{ok', ik':ik, \Gamma'_{i-1}}(q'_{i-1}) \bowtie_{ok'=ok} \pi_{ok, ik'':ik, \Gamma_i}(q_i)) \\
q'_i \equiv \pi_{ok', ik, \Gamma'_i}(q''_i)
\end{cases} \\
\hline
(\Gamma, q), Tdqd, Tq \vdash FROMe_1, \dots, e_n \Rightarrow (\Gamma'_n, @_{s:1}(\pi_{ok:ok', ik, \Gamma'_n}(q'_n))), Tq_n
\end{array}$$

**Equation 6:** FROM Rule.

As you can see the ordering, introduced by those Rules called by the FROM Rule, is retained by applying the  $\vec{\#}$  operator iteratively on the *inner keys* ( $ik'$  and  $ik''$ ) of the tabular representations produced by those Rules. The resulting column becomes the new *inner key* ( $ik$ ).

Figure 15 shows the resulting algebra plan for query  $Q1$  with the marking given in Figure 13 using these adjusted Rules. As already mentioned at the end of the compilation all the collected plans are serialized. Onto these serialized plans the two *Pathfinder* tools *pfopt* and *pfsql* are applied. The output are the resulting SQL queries. Figure 16 and 17 show the resulting queries of the markings in Figure 13. The next step is to execute these queries on the database host. Figure 18 and 19 show the resulting tables when executing the queries on the database host with the tables of Figure 3. As you can see the resulting tables correspond exactly to the tabular expressions of the marked parts of the base query.

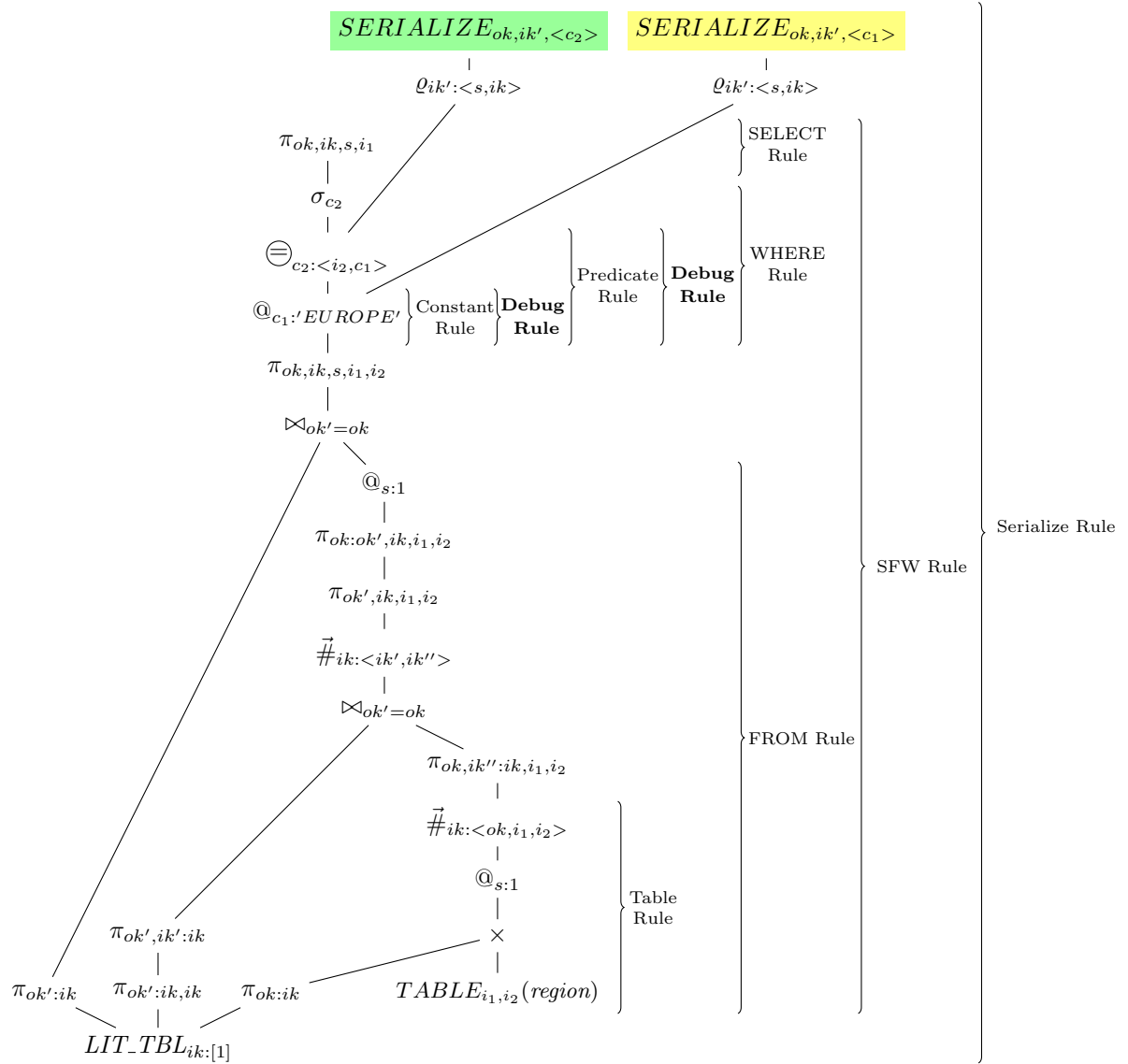


Figure 15: Algebraic code of the *Pathfinder Algebra* corresponding to query  $Q1$  with the markings given in Figure 13 created by the compilation Rules in Appendix A and the adjustments given in Appendix B.

```
SELECT 'EUROPE' AS item8_str
FROM region AS a0000
ORDER BY a0000.r_name ASC, a0000.r_regionkey ASC;
```

Figure 16: Resulting query of the first marking in Figure 13

```
SELECT CASE WHEN a0000.r_name = 'EUROPE' THEN 1 ELSE 0 END AS item5_bool
FROM region AS a0000
ORDER BY a0000.r_name ASC, a0000.r_regionkey ASC;
```

Figure 17: Resulting query of the second marking in Figure 13

item8_str
EUROPE
EUROPE
EUROPE
EUROPE

Figure 18: Resulting table when executing query in Figure 16

item5_bool
false
false
false
false

Figure 19: Resulting table when executing query in Figure 17

## 2.7 Visualisation of resulting tables

After the marked expressions are compiled into working SQL queries and these queries are executed on the database host, the resulting tables need to be presented to the user in a clear visualization. The obvious idea is to show one table for every observation. But if lots of observations are made this representation can become very complex and confusing. Because many observations are related in one way or another the implication is to present them in relation to each other, depending on the different scopes the observations are made, by joining them together on their inner or outer key (*ik* or *ok*). To accomplish that the *ik* and *ok* columns are added to the output by changing the the Serialize Rule to the following.

$(\{\}, LIT\_TBL_{ik:[1]}), \{\}, \{\} \vdash e \Rightarrow (\Gamma', q'), Tq$ $Tq : \{(\Gamma_{D_1}, q_{D_1}), \dots, (\Gamma_{D_n}, q_{D_n})\}$ $\forall i \in (1, \dots, m) \left\{ q'_{D_i} \equiv SERIALIZE_{ok, ik', <ok, ik, \Gamma_{D_i}>} (\varrho_{ik': <s, ik>} q_{D_i}) \right.$ <hr style="border: 0.5px solid black;"/> $\vdash e \Rightarrow^+ (\Gamma_{D_1}, q'_{D_1}), \dots, (\Gamma_{D_m}, q'_{D_m})$
--

**Equation 7:** Modified Serialize Rule.

If you compare it to the Serialize Rule in Appendix B you can see that the *ok* and *ik* columns are added to the *Serialize* operator and therefore to the output. After the necessary joins, that are needed in order to join the different observations together, are done all *ok* and *ik* columns are made invisible in the concrete visualization.

To display the relationship of the observations correctly we can distinguish three cases:

- **The observations are on the same scope**

This case is the easiest. To show the observations in relation to each other the inner keys (*ik*) need to be joined together. This is done via a full outer join in order to keep the discarded entries present as *null* values. Because this join is done on the GUI level, these values can be differentiated from normal *null* values and displayed accordingly (e.g. grayed out).

- **The observations are on related scope**

If two observations are on two directly connected scopes, they can be joined together on the *outer key* (*ok*) of the inner scope and *inner key* (*ik*) of the outer scope. If the observations are spanning more than two scopes, the scopes between them need to be considered. Therefore all those scopes between

this two observations are joined together in order to display the relationship correctly.

- **The observations are on unrelated scopes**

The most common ancestor scope needs to be found and joined via the previously described way iteratively onto both scopes. The observations are now on the same level and can be displayed in relation. If the most common ancestor scope is the most outer SFW block it will be ignored and the observations are considered unrelated (otherwise all observations would be related).

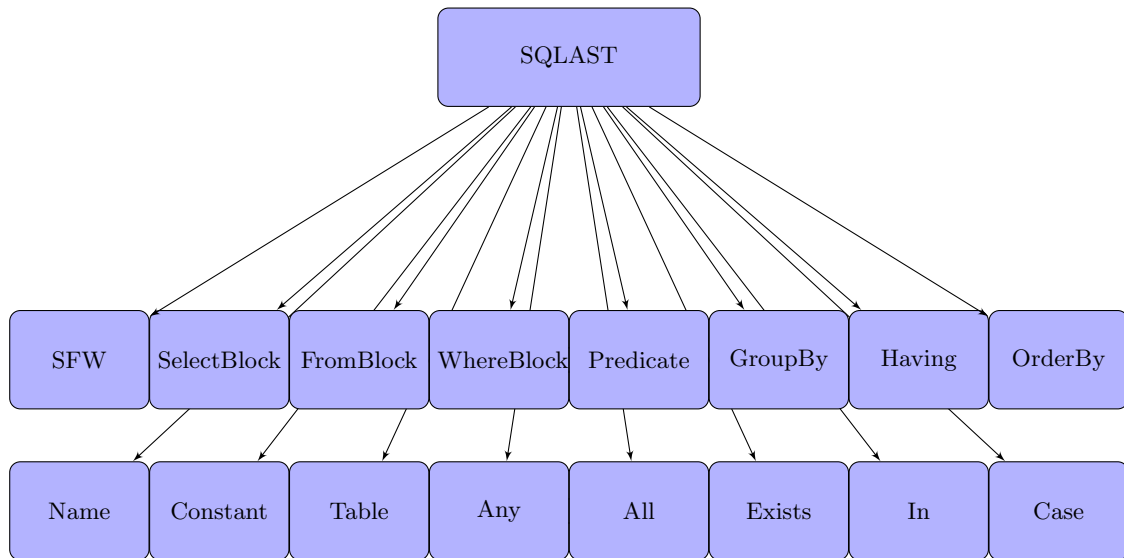


Figure 20: Hierarchical Java class-structure of the SQL AST components

## 2.8 Implementation in Java

The first thing to implement is a GUI that offers the user the ability to type in the SQL query, place the markings and start the debugging process. To implement this, the *Swing* API is used. The textfield is a own class that extends *JTextPane*. The markings are done via simple selection of subtext of the SQL query. The markings are implement with the *Highlighter* class of swing. If you don't mark a complete subexpression, the marking is automatically extended to the next outer subexpression. This is accomplished by parsing the SQL query into its AST representation and storing the start- and endpositions of every subexpression.

### 2.8.1 The AST

The AST components are implemented as *Java* classes (for every AST component a related class is present). The classes, corresponding to the components, all extend the *SQLAST* class. Figure 20 shows the AST class hierarchy and Table 2 shows the Java AST representation in relation to the corresponding SQL expressions.

<b>SQL expressions</b>	<b>Java AST Representation</b>
SQL query	SFW
FROM Block	FromBlock
FROM Item	SQLAST
SELECT Block	SelectBlock
SELECT Item	SQLAST
WHERE Expression	WhereBlock
Subexpression	Predicate, All, Any, In, Exists
Constant (like 4,2.1,"string",...)	Constant
Column	Name
Table	Table
Case	Case
Alias	String
Schema	String
Group By	GroupBy
Having	Having
Order By	OrderBy
Update	n.a.
Delete	n.a.
Insert	n.a.

Table 2: SQL AST Java representation compared to SQL expressions

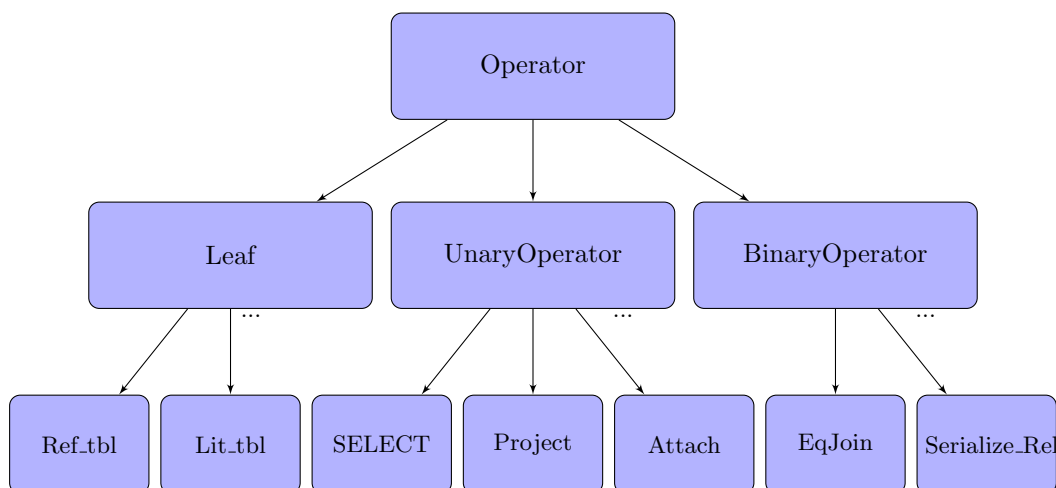


Figure 21: *Pathfinder* Operators class hierarchy

### 2.8.2 Class representation of the *Pathfinder Algebra*

The algebraic operators of the *Pathfinder Algebra* are also represented as Java classes. This class representation is necessary to accomplish the compilation of the SQL AST into the *Pathfinder Algebra*. For every algebraic operator a corresponding Java class is present. It is an intermediate stage to simplify and unitize the compilation. These Java classes store information of the various operators like kind of the operator, referenced tables and columns, child operators, newly created columns and so on. Instances of these classes can, of course, not execute the algebraic operations themselves, but the whole representation of such a algebraic query can be parsed into a XML representation that can, in turn, be handled by the *Pathfinder* tools *pfopt* and *pfsql*. The common superclass of all operator classes is the *Operator*-class, which contains basic properties of all operators. There are three subclasses of the *Operator*-class. The *BinaryOperator*-class and *UnaryOperator* for non Leaf Operators and the *Leaf*-class for Leaf Operators (operators without child operators). The particular operator classes like *Projection*, *Attach*, *EqJoin* and so on are extending the *Leaf*, *UnaryOperator* or *BinaryOperator*-class and represent the specific *Pathfinder* operators. Figure 21 shows the hierarchical structure of these classes. These classes can be instantiated by passing all information that is needed by the particular operator to the constructor. Furthermore one needs to pass references to the child operator classes on non leaf nodes. The Visitor Design Pattern is used to convert algebraic queries from the Java representation into the XML representation. Once all operator instances of a query are created you can call the *DAGaccept()* function on the root operator by passing an instance of the *XMLCreator* class that will parse the Java representation into the XML representation.

### 2.8.3 The compilation Rules

The compilation Rules are also implemented as Java classes. For every Rule a Java class is present, containing a static method *compile*. All Rules extend the *Rule* class. This class is also responsible for calling the specific Rules. If a Rule wants to start a new Rule, it first calls the generic *compileTable* or *compileTable* methods of the *Rule* class. This method then calls the appropriate Rule depending on the given AST parameter and so on. Program 1 shows an excerpt of the implementation of the FROM Rule in Java. If you compare it to Equation 27, you can see how the different parts of the Rule are implemented.

### 2.8.4 Collecting and converting the resulting plans

After the Rules are done, the resulting plans need to be serialized in order to be handled by the *Pathfinder* tools *pfopt* and *psql*. This is done by the *DebugPlansCollector* class. This class completes the compilation process by calling the *Serialize Rule* for every plan. After that, the Java representation of the plans is parsed into a XML representation and all plans are aggregated into one XML plan bundle. The next step is the execution of *pfopt* and *psql*. This is done by the *AlgebraToSQL* class. With help of the *ProcessBuilder* of the *java.lang* package the two tools are executed and applied to the plan bundle (which is temporary stored into a file). The output is a new XML plan bundle that contains the resulting SQL queries.

### 2.8.5 Extraction of SQL queries and execution with jdbc

These SQL queries need to be extracted from the XML plan. This is done by the *XMLImporter* class, that uses XPath to extract the queries.

After the queries are extracted, they can be executed on the host database. To do that, *jdbc* is used. The *DBHandler* class reads the needed database access configurations from a config file and connects to the database. After that, every SQL query is executed on the host database and the result is stored in a *ResultSet* Java object. For every *ResultSet* object a corresponding *ResultSetModel* object (provided by the *Jide* API) is constructed. The models of relating subexpressions are aggregated into a new model called *ResultModel*. This class is also responsible for joining the related results correctly together (see Section 2.7).

```

// (1)
T_[0]      = new Environment();
// (2)
q_[0]      = new Project(
                q,
                new Column[]{ok_()},
                new Column[]{ik_()});
// (3)
for (int i = 1; i <= n; i++) {
    // (3.1)
    EnvironmentOperatorPair Tiqi = compileTable(Tq,tdqd, e1_n.get(i-1));
    Environment Ti = Tiqi.env;
    Operator qi = Tiqi.op;
    // (3.2)
    T_[i] = T_[i-1].union(Ti);
    // (3.3)
    q_[i] = new Project(
        new EqJoin(
            new Project(
                q_[i-1],
                ArrayOps.union(new Column[]{ok_()},getIx(T_[i-1]))),
            new Project(
                qi,
                ArrayOps.union(new Column[]{ok_()},getIx(Ti))),
            ok_(),
            ok()),
        ArrayOps.union(ArrayOps.union(
            new Column[]{ok_()},getIx(T_[i-1])),getIx(Ti)));
}
// (4)
EnvironmentOperatorPair Tn_q_ = new EnvironmentOperatorPair(
    T_[n],
    new Rowid(
        new Attach(
            new Project(
                q_[n],
                ArrayOps.union(new Column[]{ok_() }, getIx(T_[n])),
                ArrayOps.union(new Column[]{ok_() }, getIx(T_[n]))),
            "int",
            "1"),
        ik());,
    Tq,
    true);
return Tn_q_;

```

Program 1: FROM Rule in Java.

```

SELECT  n_name
FROM    nation
WHERE   EXISTS
        (SELECT r_name
         FROM  region
         WHERE r_regionkey = n_regionkey
              AND r_name = 'EUROPE');

```

Figure 22: Marking of two subexpressions of query  $Q2$

n_name	n_regionkey	r_regionkey	r_name
Argentina	1	0	Africa
Argentina	1	1	America
Argentina	1	2	Asia
Argentina	1	3	Europe
Germany	3	0	Africa
Germany	3	1	America
Germany	3	2	Asia
Germany	3	3	Europe
Japan	2	0	Africa
Japan	2	1	America
Japan	2	2	Asia
Japan	2	3	Europe

Figure 23: Two observations of query  $Q2$  shown in relation to each other with complete unrolled entries.

### 2.8.6 Visualization of the resulting tables using the *Jide* Framework

After the models are correctly constructed, the results are able to be visualized. For every *ResultModel* a *JideTable* is created and presented to the user in a new Window (*JFrame*). Relating tables are presented in the same table but with different background colors, corresponding to the color of the marked subexpression. The *Jide* Framework also offers the ability to present table-entries, that span more than one row (or column). This makes it possible to show the unrolled entries of outer scopes as aggregated rows. Figure 23 and 24 show the two related observations of the markings given in Figure 22 of query  $Q2$  with completely unrolled (Figure 23) and aggregated (Figure 24) entries of the first observation.

This chapter explained how the debugging process works and its implementation in Java. The next chapter deals with the application of the Debugger and shows how you can use it for daily work.

n_name	n_regionkey	r_regionkey	r_name
Argentina	1	0	Africa
		1	America
		2	Asia
		3	Europe
Germany	3	0	Africa
		1	America
		2	Asia
		3	Europe
Japan	2	0	Africa
		1	America
		2	Asia
		3	Europe

Figure 24: Two observations of query  $Q2$  shown in relation to each other with entries of the first observation shown aggregated.

```

SELECT  r_regionkey
FROM    region
WHERE   r_name = 'EUROPE';

```

Figure 25: query *Q1*

region	
r_regionkey	r_name
0	Africa
1	America
2	Asia
3	Europe

Figure 26: The *region* table used in the example

### 3 How the Debugger works

This Chapter will show briefly how the Debugger works. With an introductory example the main functionality and concepts are shown.

#### 3.1 An introductory example

Recall the SQL query *Q1* (Figure 25) from Chapter 1. This query is the foundation to show how the Debugger works. The query is quite simple. It operates on the *region* table shown in Figure 26. The result should be the regionkey of Europe.

After the Debugger is started, the query can be typed in or loaded from a file (via *CMD + O*). After this is done, one can go to mark mode with *CMD + T*. Here different parts of the query can be marked for observation by selecting them and clicking *CMD + M*. With *CMD + E* the selected part can be extended to the surrounding part. The maximum extended part is the most outer SFW block. This observation of query *Q1* is shown in Figure 27. The debugging process of the marked part is started by clicking *CMD+D*. The result is the table shown in Figure 28. You can see that the table is empty.

```
SELECT r_regionkey
FROM region
WHERE r_name = 'EUROPE';
```

Figure 27: Marking of the whole query

r_regionkey	r_name

Figure 28: Resulting table when the whole query is observed

```
SELECT r_regionkey
FROM region
WHERE r_name = 'EUROPE';
```

Figure 29: Marking of the region table

r_regionkey	r_name
0	Africa
1	America
2	Asia
3	Europe

Figure 30: Resulting table when the region table is observed

Figure 29 and 30 show what happens if you observe the region table. The result should be clear.

There are a few configurations one can make. By clicking *CMD +*, you can access the preferences window. On the bottom there are two buttons to change the Editor and Table font. You can click on it and select your favorite font. On the top you can enable or disable a checkbox named *show markings in relation*. The checkbox changes the way that marked expressions are presented. If this checkbox is disabled, the observation is completely independent and you can see exactly what the result of this observation at a specific time is. If the checkbox is enabled the observations are shown in relation to other observations that might be present and to the most outer scope. This means for example that discarded entries are shown as values that are grayed out. Figure 31 and 32 show what happens if you observe the whole query and the FROM part with this option enabled.

As you can see the rows that are discarded after the evaluation of the WHERE part are grayed out. This offers the possibility to see exactly where specific values are discarded and therefore makes it easy to understand the execution process of the query. In our example every row gets grayed out so obviously the WHERE

```

SELECT  r_regionkey
FROM    region
WHERE   r_name = 'EUROPE';

```

Figure 31: Marking of the whole query and the FROM part

r_regionkey	r_name	r_regionkey
0	Africa	
1	America	
2	Asia	
3	Europe	

Figure 32: Resulting table when the whole query and the FROM part are observed

block discards everything.

If the checkbox is disabled the observation simply results in two independent result tables. These tables are shown in two different tabs. With the decouple button you can decouple a tab to get a separate window. You can also take a more detailed look at the query execution to find out why every row gets discarded. Figure 33 and Figure 34 show observations of the predicate in the WHERE part. You can see how the predicate is evaluated. Because no *r\_name* row equals 'EUROPE' the predicate evaluates completely to *false*. After the WHERE part these rows get discarded.

The obvious conclusion is that 'EUROPE' was spelled wrong. So the query gets changed to query *Q3* shown in figure 35.

Again we make an observation of the predicate and additionally observe the overall result. Figure 41 and Figure 37 show these observations. As you can see the predicate is no longer evaluated to false completely. The last row of the table survives because the entry in the *r\_name* column entry equals 'Europe'. The SELECT clause of the query then projects onto the corresponding regionkey column.

```

SELECT  r_regionkey
FROM    region
WHERE   r_name = 'EUROPE';

```

Figure 33: Marking of the predicate in the WHERE part

r_name	'EUROPE'	r_name = 'EUROPE'
Africa	EUROPE	false
America	EUROPE	false
Asia	EUROPE	false
Europe	EUROPE	false

Figure 34: Resulting table when predicates in the WHERE part are observed

```
SELECT r_regionkey
FROM region
WHERE r_name = 'Europe';
```

Figure 35: query  $Q3$

```
SELECT r_regionkey
FROM region
WHERE r_name = 'Europe';
```

Figure 36: Marking of the whole query and predicate in the WHERE part

r_name	'Europe'	r_name = 'EUROPE'	r_regionkey
Africa	Europe	false	
America	Europe	false	
Asia	Europe	false	
Europe	Europe	true	3

Figure 37: Resulting table when the whole query and predicates in the WHERE part are observed

```

SELECT  n_name
FROM    nation
WHERE   EXISTS
        (SELECT r_name
         FROM  region
         WHERE r_regionkey = n_regionkey
              AND r_name = 'EUROPE');

```

Figure 38: Query *Q2*

nation		region	
n_name	n_regionkey	r_regionkey	r_name
Argentina	1	0	Africa
Germany	3	1	America
Japan	2	2	Asia
		3	Europe

Figure 39: The tables used in the example: *nation* and *region*

### 3.2 Advanced observations

So far the base query was very simple. The following will show advanced observations with a query that is slightly more complicated and operates on different scopes. query *Q2* in Figure 38 (already used in Chapter 2) is operating on the tables *nation* and *region* shown in Figure 39). The result should be the names of the countries in the *nation* table that belong to Europe.

First one can take a look at the result of the query by observing the whole query. Figure 40 shows the result (with the *show markings in relation* checkbox disabled). Again, we can see that the table is empty.

r_regionkey	r_name

Figure 40: Resulting table when the whole query is observed

To discover what went wrong we first want take a look at the results of the subquery in relation to the outer scope. Therefore we enable the *show markings in relation* checkbox and observe the FROM clause of the top-level SFW clause in relation to the subquery FROM clause and the overall result of that subquery. Figure 41 shows the markings and Figure 42 the resulting tables. As we can see the subquery yields no results. We can conclude that the bug is located in the subquery WHERE clause.

```

SELECT  n_name
FROM    nation
WHERE   EXISTS
        (SELECT  r_name
         FROM    region
         WHERE   r_regionkey = n_regionkey
         AND    r_name = 'EUROPE')

```

Figure 41: Query  $Q_2$  with markings

n_name	n_regionkey	r_regionkey	r_name	r_name
Argentina	1	0	Africa	
		1	America	
		2	Asia	
		3	Europe	
Germany	3	0	Africa	
		1	America	
		2	Asia	
		3	Europe	
Japan	2	0	Africa	
		1	America	
		2	Asia	
		3	Europe	

Figure 42: Resulting table when the outer FROM clause and the subquery are observed

```

SELECT  n_name
FROM    nation
WHERE   EXISTS
        (SELECT  r_name
         FROM    region
         WHERE   r_regionkey = n_regionkey
         AND    r_name = 'EUROPE' )

```

Figure 43: Query  $Q_2$  with markings

n_regionkey	r_regionkey	n_regionkey = r_regionkey	r_name	'EUROPE'	r_name = 'EUROPE'
1	0	false	Africa	EUROPE	false
1	1	true	America	EUROPE	false
1	2	false	Asia	EUROPE	false
1	3	false	Europe	EUROPE	false
3	0	false	Africa	EUROPE	false
3	1	false	America	EUROPE	false
3	2	false	Asia	EUROPE	false
3	3	true	Europe	EUROPE	false
2	0	false	Africa	EUROPE	false
2	1	false	America	EUROPE	false
2	2	true	Asia	EUROPE	false
2	3	false	Europe	EUROPE	false

Figure 44: Resulting table when the predicates in the WHERE clause of the subquery are observed

Therefore we observe the two equal-predicates in the subquery. Figure 43 shows the markings and Figure 44 the resulting tables. As we can see the first predicate is evaluated correctly. The second on the other side always evaluates to false because no  $r\_name$  row equals 'EUROPE'. The obvious conclusion is that 'Europe' must be used instead of 'EUROPE'.

## 4 Performance Analysis

In the previous Chapters we have described the internal details of the SQL Debugger and how it works. So far, the performance of the compilation process and the execution of the resulting SQL queries has been ignored. This Chapter deals with these aspects. The following will analyze the efficiency of the SQL Debugger by measuring and comparing different queries. We will use the business oriented queries of TPC-H, an ad-hoc, decision supported benchmark ([4]), as basis to measure the compilation- and execution time of observations. Those queries are chosen because they have broad industry-wide relevance. All those queries are given in Appendix C. The time is measured in Java by taking and comparing the time before and after the different actions (SQL to SQL compilation, execution of the base query and execution of the resulting query) are applied. At first the time of the compilation of the base SQL query into the resulting queries is measured. This contains the parsing process of the SQL query into the SQL AST, the compilation of the SQL AST into the *Pathfinder Algebra* representation in Java, the parsing of this Java representation into the XML representation, the optimization of this XML representation and the compilation back into a SQL query.

After that, the second measuring process will compare the execution time of the compiled queries to the execution time of the base query.

To simplify the measuring processes we always consider only one observation, where the whole query is marked. This method is chosen because the measured time of observed subexpressions is nearly the same in most cases and often less. Only exotic observation, like those observing cross products of huge tables or the like, are an exception. Also the measured time can rise somewhat if a huge amount of observations are made at the same time. All in all the observation of the whole query gives a good overview over the default compilation and execution time of observations applied to a specific query.

### 4.1 SQL to SQL compilation time

At first the SQL to SQL compilation time is measured. Like already mentioned the measured time includes the process of parsing the SQL query into the AST representation, compiling the AST into the *Pathfinder Algebra* Java representation, parsing this Java representation into a XML *Pathfinder Algebra* plan, optimizing the *Pathfinder Algebra* plan and compiling it back into a SQL query.

Table 3 shows the results. As you can see the measured time is almost always less than one second. On average the compilation process is therefore very efficient. Only the compilation of query *q19* (shown in Figure 65) is out of line. This is caused by the huge amount of equations in this query.

The compilation time is also independent of the database size and therefore doesn't carry much weight in most use cases. We can conclude that the Debugger offers the ability to compile even very complex observations in short time into new SQL

Query	Compilation Time (ms)
q1	544
q2	1,235
q3	776
q4	219
q5	313
q6	277
q7	611
q8	779
q9	316
q10	335
q11	376
q12	1,148
q13	189
q14	327
q15	198
q16	1,057
q17	384
q18	664
q19	23,982
q20	705
q21	1,110
q22	1,467

Table 3: SQL to SQL compilation time.

queries. This efficiency lays the groundwork for a proper and usable application of the Debugger because these measured times strongly depend on the core implementation of the Debugger. The execution times, that are measured next, on the other hand also depend on further aspects besides the compilation Rules, like the implementation of the *Pathfinder* tools *pfopt* and *pfsql* and the underlying database host.

## 4.2 Execution time of the resulting SQL code

We measure the execution time of the resulting query in relation to the execution time of the base query. Again the TPC-H queries, given in Appendix C, are used. The underlying dbms is *db2* and the database has the sizes of 0.1 GB and 1GB. The option *'show markings in relation'* (see Chapter 3) is disabled and the strict ordering (see Section 2.6) is turned off. Table 4 and Figure 45 show the results of the 0.1GB database. Table 5 and Figure 46 show the results of the 1GB database.

Query	Original query execution time (s)	Resulting query execution time (s)
q1	3.50	3.65
q2	0.187	> 60
q3	0.371	0.389
q4	0.430	0.328
q5	0.324	0.470
q6	0.346	0.445
q7	0.416	0.537
q8	0.378	0.855
q9	1.13	1.08
q10	1.14	1.30
q11	0.810	> 60
q12	0.351	> 60
q13	0.397	0.528
q14	0.323	0.553
q15	0.191	7.33
q16	0.137	> 60
q17	2.28	> 60
q18	1.12	> 60
q19	3.14	> 60
q20	13.1	> 60
q21	0.574	> 60
q22	0.797	2.87

Table 4: SQL query timings (wall-clock execution time, in seconds) (100 MB)

Query	Original query execution time (s)	Resulting query execution time (s)
q1	30.8	32.4
q2	7.14	> 600
q3	3.69	3.74
q4	44.1	45.9
q5	6.97	8.00
q6	2.55	2.31
q7	8.01	8.00
q8	3.40	10.1
q9	13.0	14.4
q10	13.2	16.0
q11	4.76	> 600
q12	16.5	> 600
q13	3.78	5.61
q14	1.67	6.17
q15	2.53	> 600
q16	2.50	> 600
q17	57	> 600
q18	15.8	> 600
q19	21.7	> 600
q20	157	> 600
q21	19.2	> 600
q22	4.31	12.7

Table 5: SQL query timings (wall-clock execution time, in seconds) (1 GB)

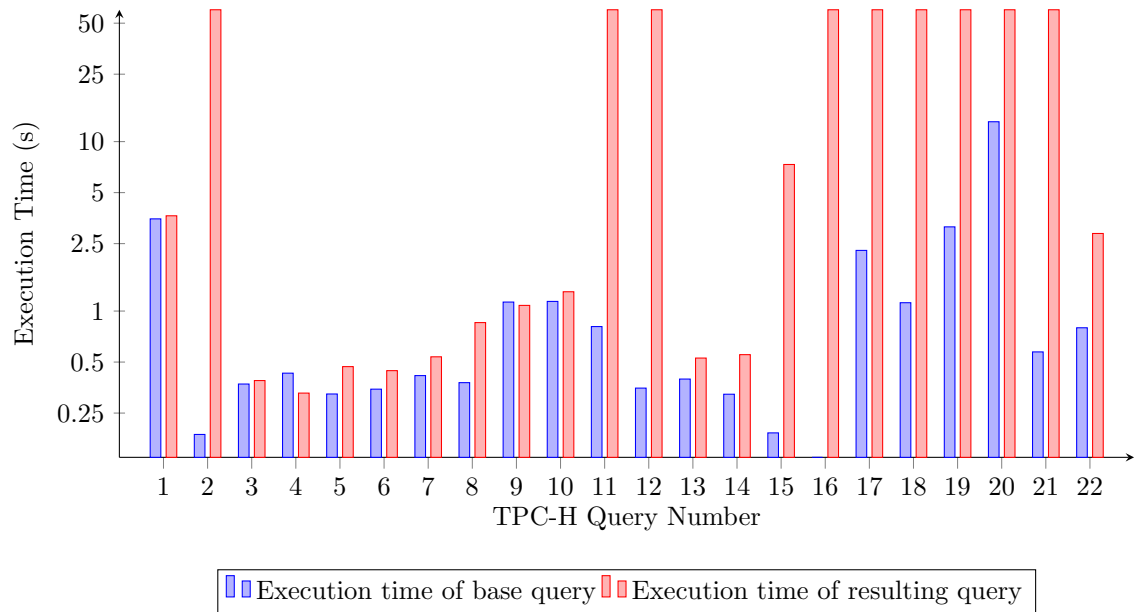


Figure 45: Bar chart of SQL execution time on a 100 MB database instance (bars that are drawn through denote a execution time bigger than 60 seconds)

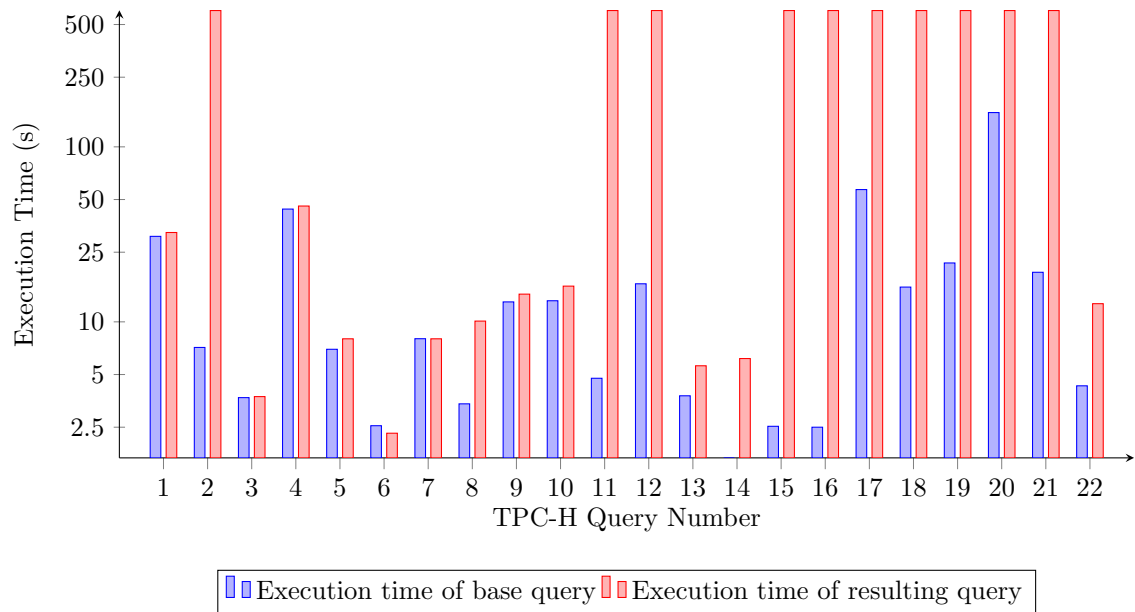


Figure 46: Bar chart of SQL execution time on a 1 GB database instance (bars that are drawn through denote a execution time bigger than 600 seconds)

As you can see lots of the queries have similar execution time as the base queries. But other queries still have performance issues. Appendix C shows that most of the problematic queries have subqueries referenced by a predicate ( $q2, q11, q17, q20$ ) or by an In or Exists expression ( $q16, q18, q20, q21$ ). Often compared with a Aggregation without a GroupBy ( $q2, q11, q17, q20$ ). The created SQL queries contain unneeded cross products that lead to this performance. These issues aren't major problems and shall be fixed with future releases of *Pathfinder* and/or improvements in the compilation process.

All in all the measured times show that it is possible to debug even complex queries without a significant performance loss. A few resulting queries perform even better than the base queries. This becomes possible due to the optimizations made by the *Pathfinder* tool *pfopt*. Further work on the Debugger and *Pathfinder* will show that this efficiency is adaptable for all kinds of queries.

## 5 Recapitulation

### 5.1 Summary

Query writing can become very tricky. You often face situations where bugs are present and you have to choose between different methods to hunt it down. As we have seen the SQL Debugger is a very useful and valuable tool that can help finding all kinds of bugs in SQL queries very quick and simple. Especially for people who's daily work is query writing, the Debugger can save a lot of time and effort. Instead of rewriting flawed SQL queries in a very annoying and exhausting process by hand, in order to hunt down the bug, the Debugger completely automates this process and presents the tabular results of interesting expressions. These results are created directly in the used database host itself. The Debugger is very easy to use and offers a wide support of the SQL language. The application field is therefore already very huge. It can be used in all kinds of industrial processes where databases are present. Also it is very useful for teaching and research purposes in the database world. It can teach students very clearly and vivid how different queries are evaluated step by step and can be used to analyze and disassemble all kinds of query in very detail. Another important aspect is that an approach like this, where you can mark subexpression on language-level and don't need to care about the underlying algebraic structures, is unique. Something like that has not been done before and it changes the way of debugging SQL queries.

### 5.2 Prospect

Although this project is already usable and can help query writers in many cases, a lot of work still needs to be done. As we have seen in Chapter 4, while the performance of the compilation process is already very good, the execution time of the resulting queries has still a lot of potential of improvement. Refinements of specific compilation Rules and adjustments to the *Pathfinder* plan optimizer are logical consequences that can help to achieve these improvements. Also the support of SQL expressions is not complete. There are still some aspects of SQL, like recursion, that can't be handled by the Debugger right now. The creation of new Rules for unsupported SQL expression can further increase the amount of supported SQL expressions. Therefore further research in this area is vital in order to improve the Debugger and widen it's application field. The groundwork to a efficiently and universally usable Debugger has already been layed. Future enhancements and improvements will widen the application range and optimize the efficiency even further.

## A Compilation Rules

### Data structures appearing in the Rules:

- $\Gamma \equiv \{col_1 \rightarrow c_1, \dots, col_n \rightarrow c_n\}$

The *Environment* ( $\Gamma$ ) contains references to the currently visible algebraic columns ( $c_1, \dots, c_n$ ) and the corresponding SQL columns ( $col_1, \dots, col_n$ ).

- $q$

The *Operator* ( $q$ ) collects all the algebraic operators and contains the resulting algebraic query at the end of the compilation process.

- $Tdqd \equiv \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_n, q_n), Tdqd_n\}$

This nested structure is used in order to realize correct compilation Rules for the GroupBy statement in SQL. A *Tdqd* structure contains a list of *Environment-Operator* pairs and further *Tdqd* structures corresponding to those *Environment-Operator* pairs.

### Functions appearing in the Rules:

- $min_i\{(col_i \rightarrow c_i) \in \Gamma\}, min_i\{(\Gamma_i, q_i), Tdqd_i \in Tdqd\}$

Returns the element with minimal index  $i$  in  $\Gamma/Tdqd$ .

- $Vars(e)$

The function  $Vars(e)$  returns all *row variables* occurring in expression  $e$  except those encapsulated by an Aggregation

- $i_{New()}$

Creates a new, unique algebraic column name.

### Compilation of GroupBy statements:

At the beginning of the compilation the initial *Tdqd* structure is empty. Whenever a GroupBy statement (or an Aggregation without a GroupBy statement) is present it is filled. The compilation process then adds the  $(\Gamma, q)$  pair (and its *Tdqd* structure) that corresponds to the unaggregated state of the query at the very front of the current *Tdqd* structure. The compilation then continues with the  $(\Gamma, q)$  pair of the aggregated state. Whenever an SQL Aggregation ( $MAX, MIN, COUNT, \dots$ ) is present the current *Tdqd* structure is accessed. The first  $(\Gamma, q)$  pair where the  $\Gamma$  corresponds to the columns in the Aggregation is used for the further compilation of the subexpression encapsulated by the Aggregation. This way the nearest valid scope is used and the compilation behaves correctly.

$$\begin{array}{l}
F \left\{ \begin{array}{l}
(\Gamma, q), \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\} \vdash FROM e_F \Rightarrow (\Gamma_F, q_F) \\
\Gamma'_F \equiv \Gamma_F + \Gamma \\
q'_F \equiv \pi_{ok, ik, s, \Gamma'_F}(q_F \bowtie_{ok=ok'} (\pi_{ok': ik, \Gamma}(q))) \\
q_{f_1} \equiv \pi_{ok: ik', ik, s, \Gamma_1}((\pi_{ok': ok, ik': ik}(q_F)) \bowtie_{ok'=ok} (\pi_{ok, ik, s, \Gamma_1}(q_1))) \\
\vdots \\
q_{f_m} \equiv \pi_{ok: ik', ik, s, \Gamma_m}((\pi_{ok': ok, ik': ik}(q_F)) \bowtie_{ok'=ok} (\pi_{ok, ik, s, \Gamma_m}(q_m)))
\end{array} \right. \\
W \left\{ (\Gamma'_F, q'_F), \{(\Gamma_1, q_{f_1}), Tdqd_1, \dots, (\Gamma_m, q_{f_m}), Tdqd_m\} \vdash WHERE e_W \Rightarrow (\Gamma_W, q_W) \right. \\
G \left\{ \begin{array}{l}
\Gamma_{G_0} \equiv \{\} \\
\forall i \in (1, \dots, n) : \left\{ \begin{array}{l}
(\Gamma_W, q_W), \{\} \vdash e_{G_i} \triangleright (q_{G_i}, c_{G_i}) \\
\Gamma_{G_i} \equiv \Gamma_{G_{i-1}} + \{e_{G_i} \rightarrow c_{G_i}\}
\end{array} \right. \\
\Gamma_G \equiv \Gamma_{G_n} - \Gamma \\
q_{G_g} \equiv \varrho_{ik': <ok, \Gamma_G>}(q_W) \\
q_G \equiv \delta(@_{s:1}(\pi_{ok, ik: ik', \Gamma_G}(q_{G_g}))) \\
\Gamma_g \equiv \Gamma_W \\
q_g \equiv \pi_{ok: ik', ik, s, \Gamma_g}(q_{G_g}) \\
\Gamma'_G \equiv \Gamma_G + \Gamma \\
q'_G \equiv \pi_{ok, ik, s, \Gamma'_G}(q_G \bowtie_{ok=ok'} (\pi_{ok': ik, \Gamma}(q))) \\
q_{g_1} \equiv \pi_{ok: ik', ik, s, \Gamma_1}((\pi_{ok': ok, ik': ik}(q_G)) \bowtie_{ok'=ok} (\pi_{ok, ik, s, \Gamma_1}(q_1))) \\
\vdots \\
q_{g_m} \equiv \pi_{ok: ik', ik, s, \Gamma_m}((\pi_{ok': ok, ik': ik}(q_G)) \bowtie_{ok'=ok} (\pi_{ok, ik, s, \Gamma_m}(q_m))) \\
Tdqd_g \equiv \{(\Gamma_1, q_{f_1}), Tdqd_1, \dots, (\Gamma_m, q_{f_m}), Tdqd_m\}
\end{array} \right. \\
H \left\{ \begin{array}{l}
(\Gamma'_G, q'_G), \{(\Gamma_g, q_g), Tdqd_g, (\Gamma_1, q_{g_1}), Tdqd_1, \dots, (\Gamma_m, q_{g_m}), Tdqd_m\} \vdash e_H \triangleright (q_H, c_H) \\
q'_H \equiv \sigma_{c_H}(q_H) \\
q_h \equiv q_g \bowtie_{ok=ok'} (\pi_{ok': ik}(q'_H)) \\
q_{h_1} \equiv \pi_{ok, ik, s, \Gamma_1}((\pi_{ik': ik}(q'_H)) \bowtie_{ik'=ok} (\pi_{ok, ik, s, \Gamma_1}(q_{g_1}))) \\
\vdots \\
q_{h_m} \equiv \pi_{ok, ik, s, \Gamma_m}((\pi_{ik': ik}(q'_H)) \bowtie_{ik'=ok} (\pi_{ok, ik, s, \Gamma_m}(q_{g_m})))
\end{array} \right. \\
S \left\{ (\Gamma'_G, q'_H), \{(\Gamma_g, q_h), Tdqd_g, (\Gamma_1, q_{h_1}), Tdqd_1, \dots, (\Gamma_m, q_{h_m}), Tdqd_m\} \vdash SELECT e_S \Rightarrow (\Gamma_S, q_S) \right. \\
O \left\{ (\Gamma'_S, q_S), \{\} \vdash ORDER BY e_O \Rightarrow (\Gamma_O, q_O) \right. \\
\hline
(\Gamma, q), \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\} \vdash SELECT e_S FROM e_F WHERE e_W GROUP BY e_{G_1}, \dots, e_{G_n} HAVING e_H ORDER BY e_O \Rightarrow (\Gamma_O, q_O)
\end{array}$$

**Equation 8:** SFW Rule.

$$\begin{array}{l}
F \left\{ \begin{array}{l}
(\Gamma, q), \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\} \vdash FROM e_F \Rightarrow (\Gamma_F, q_F) \\
\Gamma'_F \equiv \Gamma_F + \Gamma \\
q'_F \equiv \pi_{ok,ik,s,\Gamma'_F}(q_F \bowtie_{ok=ok'} (\pi_{ok':ik,\Gamma}(q))) \\
q_{f_1} \equiv \pi_{ok:ik',ik,s,\Gamma_1}((\pi_{ok':ok,ik':ik}(q_F)) \bowtie_{ok'=ok} (\pi_{ok,ik,s,\Gamma_1}(q_1))) \\
\vdots \\
q_{f_m} \equiv \pi_{ok:ik',ik,s,\Gamma_m}((\pi_{ok':ok,ik':ik}(q_F)) \bowtie_{ok'=ok} (\pi_{ok,ik,s,\Gamma_m}(q_m)))
\end{array} \right. \\
\\
W \left\{ (\Gamma'_F, q'_F), \{(\Gamma_1, q_{f_1}), Tdqd_1, \dots, (\Gamma_m, q_{f_m}), Tdqd_m\} \vdash WHERE e_W \Rightarrow (\Gamma_W, q_W) \right. \\
\\
S \left\{ \begin{array}{l}
\text{IF } AGGR \in e_S \\
\text{THEN} \\
q_G \equiv @_{s:1}(\pi_{ok:ik,ik,\Gamma}(q)) \\
\Gamma_G \equiv \Gamma \\
Tdqd_g \equiv \{(\Gamma_1, q_{f_1}), Tdqd_1, \dots, (\Gamma_m, q_{f_m}), Tdqd_m\} \\
Tdqd \equiv \{(\Gamma_W, q_W), Tdqd_g, (\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\} \\
\text{ELSE} \\
q_G \equiv q_W \\
\Gamma_G \equiv \Gamma_W \\
Tdqd \equiv \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\} \\
\text{END} \\
(\Gamma_G, q_G), Tdqd \vdash SELECT e_S \Rightarrow (\Gamma_S, q_S)
\end{array} \right. \\
\\
O \left\{ (\Gamma'_S, q_S), \{\} \vdash ORDER BY e_O \Rightarrow (\Gamma_O, q_O) \right. \\
\\
\hline
(\Gamma, q), \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\} \vdash SELECT e_S FROM e_F WHERE e_W ORDER BY e_O \Rightarrow (\Gamma_O, q_O)
\end{array}$$

**Equation 9:** SFW Rule without Group By.

$$\begin{array}{l}
\Gamma'_0 \equiv \{\} \\
q'_0 \equiv \pi_{ok':ik}(q) \\
\forall i \in (1, \dots, n) : \left\{ \begin{array}{l}
(\Gamma, q), Tdqd \vdash e_i \Rightarrow (\Gamma_i, q_i) \\
\Gamma'_i \equiv \Gamma'_{i-1} + \Gamma_i \\
q'_i \equiv \pi_{ok',\Gamma'_i}(\pi_{ok',\Gamma'_{i-1}}(q'_{i-1}) \bowtie_{ok'=ok} \pi_{ok,\Gamma_i}(q_i))
\end{array} \right. \\
\\
\hline
(\Gamma, q), Tdqd \vdash FROM e_1, \dots, e_n \Rightarrow (\Gamma'_n, \#_{ik}(@_{s:1}(\pi_{ok:ok',\Gamma'_n}(q'_n)))
\end{array}$$

**Equation 10:** FROM Rule.

$$\begin{array}{l}
\Gamma_0 \equiv \{\} \\
q_0 \equiv q \\
\forall i \in (1, \dots, n) : \begin{cases} (\Gamma, q_{i-1}), Tdqd \vdash e_i \triangleright (q_i, c_i) \\ \Gamma_i \equiv \Gamma_{i-1} + \{-col_i \rightarrow c_i\} \end{cases} \\
\hline
(\Gamma, q), Tdqd \vdash SELECT e_1 AS col_1, \dots, e_n AS col_n \Rightarrow (\Gamma_n, \pi_{ok, ik, s, \Gamma_n}(q_n))
\end{array}$$

**Equation 11:** SELECT Rule.

$$\begin{array}{l}
(\Gamma, q), Tdqd \vdash e \triangleright (q', c) \\
q'' \equiv \sigma_c(q') \\
\hline
(\Gamma, q), Tdqd \vdash WHERE e \Rightarrow (\Gamma, q'')
\end{array}$$

**Equation 12:** WHERE Rule.

$$\begin{array}{l}
q_0 \equiv q \\
\forall i \in (1, \dots, n) : \begin{cases} (\Gamma, q_{i-1}), Tdqd \vdash e_i \triangleright (q_i, c_i) \\ q' \equiv \pi_{ok, ik, s: s', \Gamma}(\#_{s': < c_1: ord_1, \dots, c_n: ord_n > / ok}(q_n)) \end{cases} \\
\hline
(\Gamma, q), Tdqd \vdash ORDER BY e_1 ord_1, \dots, e_n ord_n \Rightarrow (\Gamma, q')
\end{array}$$

**Equation 13:** OrderBy Rule.

$$\begin{array}{l}
(\{\}, LIT\_TBL_{ik: [1]}, \{\}) \vdash e \Rightarrow (\Gamma', q') \\
q'' \equiv SERIALIZE_{ok, s, < \Gamma' >}(q') \\
\hline
\vdash e \Rightarrow (\Gamma', q'')
\end{array}$$

**Equation 14:** Serialize Rule.

$$\begin{array}{l}
m_1 \equiv i_{New()} \\
\vdots \\
m_n \equiv i_{New()} \\
q' \equiv \#_{ik}(@_{s:1}(\tau_{ok:ik}(q) \times TABLE_{m_1, \dots, m_n}(e))) \\
\Gamma' \equiv \{ e.col_1 \rightarrow m_1, \dots, e.col_n \rightarrow m_n \} \\
\hline
(\Gamma, q), Tdqd \vdash e(col_1, \dots, col_n) \Rightarrow (\Gamma', q')
\end{array}$$

**Equation 15:** Table Rule.

$$\begin{array}{l}
(col' \rightarrow c') \equiv \min_i \{ (t.col_i \rightarrow c_i) \in \Gamma : t.col_i \equiv v.col \} \\
\hline
(\Gamma, q), Tdqd \vdash v.col \triangleright (q, c')
\end{array}$$

**Equation 16:** Lookup Rule with fully qualified names.

$$\begin{array}{l}
(col' \rightarrow c') \equiv \min_i \{ (v.col_i \rightarrow c_i) \in \Gamma : col_i \equiv col \} \\
\hline
(\Gamma, q), Tdqd \vdash col \triangleright (q, c')
\end{array}$$

**Equation 17:** Lookup Rule without fully qualified names.

$$\begin{array}{l}
c \equiv i_{New()} \\
\hline
(\Gamma, q), Tdqd \vdash val \triangleright (@_{c:val}(q), c)
\end{array}$$

**Equation 18:** Constant Rule.

$$\begin{array}{l}
(\Gamma, q), Tdqd \vdash e \Rightarrow (\{col_1 \rightarrow c_1, \dots, col_n \rightarrow c_n\}, q') \\
\Gamma_0 \equiv \{ \} \\
\forall i \in (1, \dots, n) : \left\{ \Gamma_i \equiv \Gamma_{i-1} + \{t.col_i \rightarrow c_i\} \right. \\
\hline
(\Gamma, q), Tdqd \vdash e AS t \triangleright (\Gamma_n, q')
\end{array}$$

**Equation 19:** Rename Rule.

$$\begin{array}{l}
(\Gamma, q), Tdqd \vdash e_1 \triangleright (q_1, c_1) \\
(\Gamma, q_1), Tdqd \vdash e_2 \triangleright (q_2, c_2) \\
c \equiv i_{New()} \\
q' \equiv \odot_{c: \langle c_1, c_2 \rangle} (q_2) \\
\hline
(\Gamma, q), Tdqd \vdash e_1 \cdot e_2 \triangleright (q', c)
\end{array}$$

**Equation 20:** Predicate Rule.

$$\begin{array}{l}
(\Gamma_g, q_g), Tdqd_g \equiv \min_i \{ (\Gamma_i, q_i), Tdqd_i \in \{ (\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m \} : Vars(e) \in \Gamma_i \} \\
(\Gamma_g, q_g), Tdqd_g \vdash e \triangleright (q_A, c_A) \\
c \equiv i_{New()} \\
q' \equiv \text{AGG}_{c: \langle AGGR(c_A) \rangle / ok} (q_A) \\
q'' \equiv q' \cup (\@_{c: -} (\pi_{ok: ik}(q) \setminus \pi_{ok}(q'))) \\
q''' \equiv \pi_{ok, ik, s, \Gamma, c} (q \bowtie_{ik=ok'} (\pi_{ok': ok, c}(q''))) \\
\hline
(\Gamma, q), \{ (\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m \} \vdash \text{AGGR}(e) \triangleright (q''', c)
\end{array}$$

**Equation 21:** Aggregation Rule.

$$\begin{array}{l}
(\Gamma, q), Tdqd \vdash e \Rightarrow (\Gamma_E, q_E) \\
c \equiv i_{New()} \\
q' \equiv \delta(\pi_{ok}(q_E)) \\
q'' \equiv (\@_{c: true} (q')) \cup (\@_{c: false} (\pi_{ok: ik}(q) \setminus q')) \\
q''' \equiv \pi_{ok, ik, s, \Gamma, c} (q \bowtie_{ik=ok'} (\pi_{ok': ok, c}(q''))) \\
\hline
(\Gamma, q), Tdqd \vdash \text{EXISTS}(e) \triangleright (q''', c)
\end{array}$$

**Equation 22:** Exists Rule.

$$\begin{array}{l}
(\Gamma, q), Tdqd \vdash e_1 \triangleright (q_1, c_1) \\
(\Gamma, q), Tdqd \vdash e_2 \Rightarrow (\{c_2\}, q_2) \\
c \equiv i_{New()} \\
q_{12} \equiv \pi_{ok':ik,c_1}(q_1) \bowtie_{ok'=ok} \pi_{ok,c_2}(q_2) \\
q_{=} \equiv \sigma_c(\ominus_{c:\langle c_1, c_2 \rangle}(q_{12})) \\
q' \equiv \delta(\pi_{ok,c}(q_{=})) \\
q'' \equiv q' \cup (@_{c:false}(\pi_{ok:ik}(q) \setminus \pi_{ok}(q'))) \\
q''' \equiv \pi_{ok,ik,s,\Gamma,c}(q \bowtie_{ik=ok'}(\pi_{ok':ok,c}(q''))) \\
\hline
(\Gamma, q), Tdqd \vdash e_1 IN(e_2) \triangleright (q''', c)
\end{array}$$

**Equation 23:** In Rule.

$$\begin{aligned}
& (\Gamma, q), \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\} \vdash e_1 \triangleright (q1, c_1) \\
& q1' \equiv \sigma_{c_1}(q1) \\
& qc_{1_1} \equiv \pi_{ok, ik, s, \Gamma_1}((\pi_{ik': ik}(q1')) \bowtie_{ik'=ok} (\pi_{ok, ik, s, \Gamma_1}(q1))) \\
& \vdots \\
& qc_{1_m} \equiv \pi_{ok, ik, s, \Gamma_m}((\pi_{ik': ik}(q1')) \bowtie_{ik'=ok} (\pi_{ok, ik, s, \Gamma_m}(q_m))) \\
& (\Gamma, q1'), \{(\Gamma_1, qc_{1_1}), Tdqd_1, \dots, (\Gamma_m, qc_{1_m}), Tdqd_m\} \vdash e_2 \triangleright (q2, c_2) \\
& c \equiv i_{New}() \\
& q2' \equiv \sigma_c(\neg_{c:c_1}(q1)) \\
& qc_{2_1} \equiv \pi_{ok, ik, s, \Gamma_1}((\pi_{ik': ik}(q2')) \bowtie_{ik'=ok} (\pi_{ok, ik, s, \Gamma_1}(q1))) \\
& \vdots \\
& qc_{2_m} \equiv \pi_{ok, ik, s, \Gamma_m}((\pi_{ik': ik}(q2')) \bowtie_{ik'=ok} (\pi_{ok, ik, s, \Gamma_m}(q_m))) \\
& (\Gamma, q2'), \{(\Gamma_1, qc_{2_1}), Tdqd_1, \dots, (\Gamma_m, qc_{2_m}), Tdqd_m\} \vdash e_3 \triangleright (q3, c_3) \\
& c' \equiv i_{New}() \\
& q' \equiv \pi_{ik': ik, c': c_2}(q2) \cup \pi_{ik': ik, c': c_3}(q3) \\
& q'' \equiv \pi_{\Gamma, ok, ik, s, c'}(q \bowtie_{ik=ik'} q') \\
\hline
& (\Gamma, q), \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\} \vdash \text{CASE WHEN } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 \triangleright (q'', c')
\end{aligned}$$

**Equation 24:** Case Rule.



## B Extended Rules for Debugging

$$\begin{array}{l}
\left\{ \begin{array}{l}
(\Gamma, q), \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\}, Tq \vdash FROM e_F \Rightarrow (\Gamma_F, q_F), Tq' \\
\Gamma'_F \equiv \Gamma_F + \Gamma \\
q'_F \equiv \pi_{ok, ik, s, \Gamma'_F}(q_F \bowtie_{ok=ok'} (\pi_{ok': ik, \Gamma}(q))) \\
q_{f_1} \equiv \pi_{ok: ik', ik, s, \Gamma_1}((\pi_{ok': ok, ik': ik}(q_F)) \bowtie_{ok'=ok} (\pi_{ok, ik, s, \Gamma_1}(q_1))) \\
\vdots \\
q_{f_m} \equiv \pi_{ok: ik', ik, s, \Gamma_m}((\pi_{ok': ok, ik': ik}(q_F)) \bowtie_{ok'=ok} (\pi_{ok, ik, s, \Gamma_m}(q_m)))
\end{array} \right. \\
W \left\{ (\Gamma'_F, q'_F), \{(\Gamma_1, q_{f_1}), Tdqd_1, \dots, (\Gamma_m, q_{f_m}), Tdqd_m\}, Tq' \vdash WHERE e_W \Rightarrow (\Gamma_W, q_W), Tq'' \right. \\
\left. \left\{ \begin{array}{l}
\Gamma_{G_0} \equiv \{\} \\
Tq''' \equiv Tq'' \\
\forall i \in (1, \dots, n) : \left\{ \begin{array}{l}
(\Gamma_W, q_W), \{\}, Tq'''_{i-1} \vdash e_{G_i} \triangleright (q_{G_i}, c_{G_i}), Tq'''_i \\
\Gamma_{G_i} \equiv \Gamma_{G_{i-1}} + \{e_{G_i} \rightarrow c_{G_i}\}
\end{array} \right. \\
\Gamma_G \equiv \Gamma_{G_n} - \Gamma \\
q_{G_g} \equiv \varrho_{ik': <ok, \Gamma_G>}(q_W) \\
q_G \equiv \delta(@_{s:1}(\pi_{ok, ik: ik', \Gamma_G}(q_{G_g}))) \\
\Gamma_g \equiv \Gamma_W \\
q_g \equiv \pi_{ok: ik', ik, s, \Gamma_W}(q_{G_g}) \\
\Gamma'_G \equiv \Gamma_G + \Gamma \\
q'_G \equiv \pi_{ok, ik, s, \Gamma'_G}(q_G \bowtie_{ok=ok'} (\pi_{ok': ik, \Gamma}(q))) \\
q_{g_1} \equiv \pi_{ok: ik', ik, s, \Gamma_1}((\pi_{ok': ok, ik': ik}(q_G)) \bowtie_{ok'=ok} (\pi_{ok, ik, s, \Gamma_1}(q_1))) \\
\vdots \\
q_{g_m} \equiv \pi_{ok: ik', ik, s, \Gamma_m}((\pi_{ok': ok, ik': ik}(q_G)) \bowtie_{ok'=ok} (\pi_{ok, ik, s, \Gamma_m}(q_m))) \\
Tdqd_g \equiv \{(\Gamma_1, q_{f_1}), Tdqd_1, \dots, (\Gamma_m, q_{f_m}), Tdqd_m\}
\end{array} \right. \\
H \left\{ \begin{array}{l}
(\Gamma'_G, q'_G), \{(\Gamma_g, q_g), Tdqd_g, (\Gamma_1, q_{g_1}), Tdqd_1, \dots, (\Gamma_m, q_{g_m}), Tdqd_m\}, Tq''' \vdash e_H \triangleright (q_H, c_H), Tq'''' \\
q'_H \equiv \sigma_{c_H}(q_H) \\
q_h \equiv q_g \bowtie_{ok=ok'} (\pi_{ok': ik}(q'_H)) \\
q_{h_1} \equiv \pi_{ok, ik, s, \Gamma_1}((\pi_{ik': ik}(q'_H)) \bowtie_{ik'=ok} (\pi_{ok, ik, s, \Gamma_1}(q_{g_1}))) \\
\vdots \\
q_{h_m} \equiv \pi_{ok, ik, s, \Gamma_m}((\pi_{ik': ik}(q'_H)) \bowtie_{ik'=ok} (\pi_{ok, ik, s, \Gamma_m}(q_{g_m})))
\end{array} \right. \\
S \left\{ \begin{array}{l}
(\Gamma'_G, q'_H), \{(\Gamma_g, q_h), Tdqd_g, (\Gamma_1, q_{h_1}), Tdqd_1, \dots, (\Gamma_m, q_{h_m}), Tdqd_m\}, Tq'''' \\
\vdash SELECT e_S \Rightarrow (\Gamma_S, q_S), Tq''''''
\end{array} \right. \\
O \left\{ (\Gamma'_S, q_S), \{\}, Tq'''''' \vdash ORDER BY e_O \Rightarrow (\Gamma_O, q_O), Tq'''''''' \right. \\
\left. (\Gamma, q), \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\}, Tq \vdash SELECT e_S FROM e_F WHERE e_W \right. \\
\left. GROUP BY e_{G_1}, \dots, e_{G_n} HAVING e_H ORDER BY e_O \Rightarrow (\Gamma_O, q_O), Tq'''''''' \right.
\end{array}$$

Equation 25: SFW Rule.

$$\begin{array}{l}
F \left\{ \begin{array}{l}
(\Gamma, q), \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\}, Tq \vdash FROM e_F \Rightarrow (\Gamma_F, q_F), Tq' \\
\Gamma'_F \equiv \Gamma_F + \Gamma \\
q'_F \equiv \pi_{ok, ik, s, \Gamma'_F}(q_F \bowtie_{ok=ok'} (\pi_{ok': ik, \Gamma}(q))) \\
q_{f_1} \equiv \pi_{ok: ik', ik, s, \Gamma_1}((\pi_{ok': ok, ik': ik}(q_F)) \bowtie_{ok'=ok} (\pi_{ok, ik, s, \Gamma_1}(q_1))) \\
\vdots \\
q_{f_m} \equiv \pi_{ok: ik', ik, s, \Gamma_m}((\pi_{ok': ok, ik': ik}(q_F)) \bowtie_{ok'=ok} (\pi_{ok, ik, s, \Gamma_m}(q_m)))
\end{array} \right. \\
\\
W \left\{ (\Gamma'_F, q'_F), \{(\Gamma_1, q_{f_1}), Tdqd_1, \dots, (\Gamma_m, q_{f_m}), Tdqd_m\}, Tq' \vdash WHERE e_W \Rightarrow (\Gamma_W, q_W), Tq'' \right. \\
\\
S \left\{ \begin{array}{l}
\text{IF } AGGR \in e_S \\
\text{THEN} \\
q_G \equiv @_{s:1}(\pi_{ok: ik, ik, \Gamma}(q)) \\
\Gamma_G \equiv \Gamma \\
Tdqd_g \equiv \{(\Gamma_1, q_{f_1}), Tdqd_1, \dots, (\Gamma_m, q_{f_m}), Tdqd_m\} \\
Tdqd \equiv \{(\Gamma_W, q_W), Tdqd_g, (\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\} \\
\text{ELSE} \\
q_G \equiv q_W \\
\Gamma_G \equiv \Gamma_W \\
Tdqd \equiv \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\} \\
\text{END} \\
(\Gamma_G, q_G), Tdqd, Tq'' \vdash SELECT e_S \Rightarrow (\Gamma_S, q_S), Tq'''
\end{array} \right. \\
\\
O \left\{ (\Gamma'_S, q_S), \{\}, Tq''' \vdash ORDER BY e_O \Rightarrow (\Gamma_O, q_O), Tq'''' \right. \\
\\
\hline
(\Gamma, q), \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\}, Tq \vdash SELECT e_S FROM e_F WHERE e_W ORDER BY e_O \Rightarrow (\Gamma_O, q_O), Tq''''
\end{array}$$

**Equation 26:** SFW Rule without Group By.

$$\begin{array}{l}
\Gamma'_0 \equiv \{\} \\
q'_0 \equiv \pi_{ok':ik,ik}(q) \\
Tq_0 \equiv Tq \\
\forall i \in (1, \dots, n) : \begin{cases} (\Gamma, q), Tdqd, Tq_{i-1} \vdash e_i \Rightarrow (\Gamma_i, q_i), Tq_i \\ \Gamma'_i \equiv \Gamma'_{i-1} + \Gamma_i \\ q''_i \equiv \#_{ik:\langle ik', ik'' \rangle} (\pi_{ok', ik':ik, \Gamma'_{i-1}}(q'_{i-1}) \bowtie_{ok'=ok} \pi_{ok, ik'':ik, \Gamma_i}(q_i)) \\ q'_i \equiv \pi_{ok', ik, \Gamma'_i}(q''_i) \end{cases} \\
\hline
(\Gamma, q), Tdqd, Tq \vdash FROM e_1, \dots, e_n \Rightarrow (\Gamma'_n, @_{s:1}(\pi_{ok:ok', ik, \Gamma'_n}(q'_n))), Tq_n
\end{array}$$

**Equation 27:** FROM Rule.

$$\begin{array}{l}
\Gamma_0 \equiv \{\} \\
q_0 \equiv q \\
Tq_0 \equiv Tq \\
\forall i \in (1, \dots, n) : \begin{cases} (\Gamma, q_{i-1}), Tdqd, Tq_{i-1} \vdash e_i \triangleright (q_i, c_i), Tq_i \\ \Gamma_i \equiv \Gamma_{i-1} + \{col_i \rightarrow c_i\} \end{cases} \\
\hline
(\Gamma, q), Tdqd, Tq \vdash SELECT e_1 AS col_1, \dots, e_n AS col_n \Rightarrow (\Gamma_n, \pi_{ok, ik, s, \Gamma_n}(q_n)), Tq_n
\end{array}$$

**Equation 28:** SELECT Rule.

$$\begin{array}{l}
(\Gamma, q), Tdqd, Tq \vdash e \triangleright (q', c), Tq' \\
q'' \equiv \sigma_c(q') \\
\hline
(\Gamma, q), Tdqd, Tq \vdash WHERE e \Rightarrow (\Gamma, q''), Tq'
\end{array}$$

**Equation 29:** WHERE Rule.

$$\begin{array}{l}
q_0 \equiv q \\
Tq_0 \equiv Tq \\
\forall i \in (1, \dots, n) : \left\{ (\Gamma, q_{i-1}), Tdqd, Tq_{i-1} \vdash e_i \triangleright (q_i, c_i), Tq_i \right. \\
q' \equiv \pi_{ok, ik, s: s', \Gamma}(\vec{\#}_{s': < c_1: ord_1, \dots, c_n: ord_n > / ok}(q_n)) \\
\hline
(\Gamma, q), Tdqd, Tq \vdash ORDER\ BY\ e_1\ ord_1, \dots, e_n\ ord_n \Rightarrow (\Gamma, q'), Tq_n
\end{array}$$

**Equation 30:** OrderBy Rule.

$$\begin{array}{l}
(\{\}, LIT\_TBL_{ik:[1]}, \{\}, \{\} \vdash e \Rightarrow (\Gamma', q'), Tq \\
Tq \equiv \{(\Gamma_{D_1}, q_{D_1}), \dots, (\Gamma_{D_n}, q_{D_n})\} \\
\forall i \in (1, \dots, m) \left\{ q'_{D_i} \equiv SERIALIZE_{ok, ik', < \Gamma_{D_i} >}(q_{ik': < s, ik > q_{D_i}) \right. \\
\hline
\vdash e \Rightarrow^+ (\Gamma_{D_1}, q'_{D_1}), \dots, (\Gamma_{D_m}, q'_{D_m})
\end{array}$$

**Equation 31:** Serialize Rule.

$$\begin{array}{l}
m_1 \equiv i_{New()} \\
\vdots \\
m_n \equiv i_{New()} \\
q' \equiv \vec{\#}_{ik: < ok, Keys(e) >}(@_{s:1}(\pi_{ok: ik}(q) \times TABLE_{m_1, \dots, m_n}(e))) \\
\Gamma' \equiv \{ e.col_1 \rightarrow m_1, \dots, e.col_n \rightarrow m_n \} \\
\hline
(\Gamma, q), Tdqd, Tq \vdash e(col_1, \dots, col_n) \Rightarrow (\Gamma', q'), Tq
\end{array}$$

**Equation 32:** Table Rule.

$$\begin{array}{l}
(col' \rightarrow c') \equiv \min_i \{ (t.col_i \rightarrow c_i) \in \Gamma : t.col_i \equiv v.col \} \\
\hline
(\Gamma, q), Tdqd, Tq \vdash v.col \triangleright (q, c'), Tq
\end{array}$$

**Equation 33:** Lookup Rule with fully qualified names.

$$(col' \rightarrow c') \equiv \min_i \{ (v.col_i \rightarrow c_i) \in \Gamma : col_i \equiv col \}$$

$$(\Gamma, q), Tdqd, Tq \vdash col \triangleright (q, c'), Tq$$

**Equation 34:** Lookup Rule without fully qualified names.

$$c \equiv i_{New()}$$

$$(\Gamma, q), Tdqd, Tq \vdash val \triangleright (@_{c:val}(q), c), Tq$$

**Equation 35:** Constant Rule.

$$(\Gamma, q), Tdqd, Tq \vdash e \Rightarrow (\{col_1 \rightarrow c_1, \dots, col_n \rightarrow c_n\}, q'), Tq'$$

$$\Gamma_0 \equiv \{ \}$$

$$\forall i \in (1, \dots, n) : \{ \Gamma_i \equiv \Gamma_{i-1} + \{t.col_i \rightarrow c_i\} \}$$

$$(\Gamma, q), Tdqd, Tq \vdash e \text{ AS } t \triangleright (\Gamma_n, q'), Tq'$$

**Equation 36:** Rename Rule.

$$(\Gamma, q), Tdqd, Tq \vdash e_1 \triangleright (q_1, c_1), Tq'$$

$$(\Gamma, q_1), Tdqd, Tq' \vdash e_2 \triangleright (q_2, c_2), Tq''$$

$$c \equiv i_{New()}$$

$$q' \equiv \odot_{c:\langle c_1, c_2 \rangle}(q_2)$$

$$(\Gamma, q), Tdqd, Tq \vdash e_1 \cdot e_2 \triangleright (q', c), Tq''$$

**Equation 37:** Predicate Rule.

$$\begin{array}{l}
(\Gamma_g, q_g), Tdqd_g \equiv \min_i \{(\Gamma_i, q_i), Tdqd_i \in \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\} : Vars(e) \in \Gamma_i\} \\
(\Gamma_g, q_g), Tdqd_g, Tq \vdash e \triangleright (q_A, c_A), Tq' \\
c \equiv i_{New()} \\
q' \equiv \text{AGG}_{c: \langle \text{AGGR}(c_A) \rangle / ok} (q_A) \\
q'' \equiv q' \cup (\text{@}_{c: -} (\pi_{ok: ik}(q) \setminus \pi_{ok}(q'))) \\
q''' \equiv \pi_{ok, ik, s, \Gamma, c}(q \bowtie_{ik=ok'} (\pi_{ok': ok, c}(q''))) \\
\hline
(\Gamma, q), \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\}, Tq \vdash \text{AGGR}(e) \triangleright (q''', c), Tq'
\end{array}$$

**Equation 38:** Aggregation Rule.

$$\begin{array}{l}
(\Gamma, q), Tdqd, Tq \vdash e \Rightarrow (\Gamma_E, q_E), Tq' \\
c \equiv i_{New()} \\
q' \equiv \delta(\pi_{ok}(q_E)) \\
q'' \equiv (\text{@}_{c: true}(q')) \cup (\text{@}_{c: false} (\pi_{ok: ik}(q) \setminus q')) \\
q''' \equiv \pi_{ok, ik, s, \Gamma, c}(q \bowtie_{ik=ok'} (\pi_{ok': ok, c}(q''))) \\
\hline
(\Gamma, q), Tdqd, Tq \vdash \text{EXISTS}(e) \triangleright (q''', c), Tq'
\end{array}$$

**Equation 39:** Exists Rule.

$$\begin{array}{l}
(\Gamma, q), Tdqd, Tq \vdash e_1 \triangleright (q1, c_1), Tq' \\
(\Gamma, q), Tdqd, Tq' \vdash e_2 \Rightarrow (\{c_2\}, q2), Tq'' \\
c \equiv i_{New}() \\
q_{12} \equiv \pi_{ok':ik,c_1}(q_1) \bowtie_{ok'=ok} \pi_{ok,c_2}(q_2) \\
q_{=} \equiv \sigma_c(\bigoplus_{c:<c_1,c_2>}(q_{12})) \\
q' \equiv \delta(\pi_{ok,c}(q_{=})) \\
q'' \equiv q' \sqcup (@_{c:false}(\pi_{ok:ik}(q) \setminus \pi_{ok}(q'))) \\
q''' \equiv \pi_{ok,ik,s,\Gamma,c}(q \bowtie_{ik=ok'}(\pi_{ok':ok,c}(q''))) \\
\hline
(\Gamma, q), Tdqd, Tq \vdash e_1 IN(e_2) \triangleright (q''', c), Tq''
\end{array}$$

**Equation 40:** In Rule.

$$\begin{array}{l}
(\Gamma, q), \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\}, Tq \vdash e_1 \triangleright (q_1, c_1), Tq' \\
q1' \equiv \sigma_{c_1}(q1) \\
q_{c1_1} \equiv \pi_{ok, ik, s, \Gamma_1}((\pi_{ik': ik}(q1')) \bowtie_{ik'=ok} (\pi_{ok, ik, s, \Gamma_1}(q1))) \\
\vdots \\
q_{c1_m} \equiv \pi_{ok, ik, s, \Gamma_m}((\pi_{ik': ik}(q1')) \bowtie_{ik'=ok} (\pi_{ok, ik, s, \Gamma_m}(q_m))) \\
(\Gamma, q1'), \{(\Gamma_1, q_{c1_1}), Tdqd_1, \dots, (\Gamma_m, q_{c1_m}), Tdqd_m\}, Tq' \vdash e_2 \triangleright (q2, c_2), Tq'' \\
c \equiv i_{New()} \\
q2' \equiv \sigma_c(\neg_{c:c_1}(q1)) \\
q_{c2_1} \equiv \pi_{ok, ik, s, \Gamma_1}((\pi_{ik': ik}(q2')) \bowtie_{ik'=ok} (\pi_{ok, ik, s, \Gamma_1}(q1))) \\
\vdots \\
q_{c2_m} \equiv \pi_{ok, ik, s, \Gamma_m}((\pi_{ik': ik}(q2')) \bowtie_{ik'=ok} (\pi_{ok, ik, s, \Gamma_m}(q_m))) \\
(\Gamma, q2'), \{(\Gamma_1, q_{c2_1}), Tdqd_1, \dots, (\Gamma_m, q_{c2_m}), Tdqd_m\}, Tq'' \vdash e_3 \triangleright (q3, c_3), Tq''' \\
c' \equiv i_{New()} \\
q' \equiv \pi_{ik': ik, c': c_2}(q2) \cup \pi_{ik': ik, c': c_3}(q3) \\
q'' \equiv \pi_{\Gamma, ok, ik, s, c'}(q \bowtie_{ik=ik'} q') \\
\hline
(\Gamma, q), \{(\Gamma_1, q_1), Tdqd_1, \dots, (\Gamma_m, q_m), Tdqd_m\}, Tq \vdash \text{CASE WHEN } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 \\
\triangleright (q'', c'), Tq'''
\end{array}$$

**Equation 41:** Case Rule.

$$\begin{array}{l}
(\Gamma, q), Tdqd, Tq \vdash e \Rightarrow (\Gamma', q'), Tq' \\
Tq'' \equiv Tq' + \{(\Gamma', q')\} \\
\hline
(\Gamma, q), Tdqd, Tq \vdash \text{DEBUG } e \Rightarrow (\Gamma', q'), Tq''
\end{array}$$

**Equation 42:** Debug Table Rule.

$(\Gamma, q), Tdqd, Tq \vdash e \triangleright (q', c), Tq'$ $Tq'' \equiv Tq' + \{(\{col \rightarrow c\}, q')\}$
$(\Gamma, q), Tdqd, Tq \vdash DEBUG e \triangleright (q', c), Tq''$

**Equation 43:** Debug Column Rule.

## C TPC-H Queries

```
select
  l_returnflag,
  l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
from
  lineitem
where
  l_shipdate <= date ('1998-09-02')
group by
  l_returnflag,
  l_linestatus
order by
  l_returnflag,
  l_linestatus;
```

Figure 47: *q1*

```

select
    s_acctbal ,
    s_name ,
    n_name ,
    p_partkey ,
    p_mfgr ,
    s_address ,
    s_phone ,
    s_comment
from
    part ,
    supplier ,
    partsupp ,
    nation ,
    region
where
    p_partkey = ps_partkey
    and s_suppkey = ps_suppkey
    and p_size = 15
    and p_type like '%BRASS'
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'EUROPE'
    and ps_supplycost = (
        select
            min(ps_supplycost)
        from
            partsupp ,
            supplier ,
            nation ,
            region
        where
            p_partkey = ps_partkey
            and s_suppkey = ps_suppkey
            and s_nationkey = n_nationkey
            and n_regionkey = r_regionkey
            and r_name = 'EUROPE'
    )
order by
    s_acctbal desc ,
    n_name ,
    s_name ,
    p_partkey;

```

Figure 48: *q2*

```

select
  l_orderkey,
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  o_orderdate,
  o_shippriority
from
  customer,
  orders,
  lineitem
where
  c_mktsegment = 'BUILDING'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < date('1995-03-15')
  and l_shipdate > date('1995-03-15')
group by
  l_orderkey,
  o_orderdate,
  o_shippriority
order by
  revenue desc,
  o_orderdate;

```

Figure 49: *q3*

```

select
  o_orderpriority,
  count(*) as order_count
from
  orders
where
  o_orderdate >= date('1993-07-01')
  and o_orderdate < date('1993-10-01')
  and exists (
    select
      *
    from
      lineitem
    where
      l_orderkey = o_orderkey
      and l_commitdate < l_receiptdate
  )
group by
  o_orderpriority
order by
  o_orderpriority;

```

Figure 50: *q4*

```

select
    n_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue
from
    customer,
    orders,
    lineitem,
    supplier,
    nation,
    region
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and l_suppkey = s_suppkey
    and c_nationkey = s_nationkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'ASIA'
    and o_orderdate >= date('1994-01-01')
    and o_orderdate < date('1995-01-01')
group by
    n_name
order by
    revenue desc;

```

Figure 51: *q5*

```

select
    n_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue
from
    customer,
    orders,
    lineitem,
    supplier,
    nation,
    region
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and l_suppkey = s_suppkey
    and c_nationkey = s_nationkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'ASIA'
    and o_orderdate >= date('1994-01-01')
    and o_orderdate < date('1995-01-01')
group by
    n_name
order by
    revenue desc;
Fabi-MAC:TPCH-queries fabi$ cat q6.sql
select
    sum(l_extendedprice * l_discount) as revenue
from
    lineitem
where
    l_shipdate >= date('1994-01-01')
    and l_shipdate < date('1995-01-01')
    and l_discount between .06 - 0.01 and .06 + 0.01
    and l_quantity < 24;

```

Figure 52: *q6*

```

select
  supp_nation,
  cust_nation,
  l_year,
  sum(volume) as revenue
from
  (
    select
      n1.n_name as supp_nation,
      n2.n_name as cust_nation,
      year(l_shipdate) as l_year,
      l_extendedprice * (1 - l_discount) as volume
    from
      supplier,
      lineitem,
      orders,
      customer,
      nation n1,
      nation n2
    where
      s_suppkey = l_suppkey
      and o_orderkey = l_orderkey
      and c_custkey = o_custkey
      and s_nationkey = n1.n_nationkey
      and c_nationkey = n2.n_nationkey
      and (
        (n1.n_name = 'FRANCE' and n2.n_name = 'GERMANY')
        or (n1.n_name = 'GERMANY' and n2.n_name = 'FRANCE')
      )
      and l_shipdate between date('1995-01-01') and date('1996-12-31')
    ) as shipping
  group by
    supp_nation,
    cust_nation,
    l_year
  order by
    supp_nation,
    cust_nation,
    l_year;

```

Figure 53: *q7*

```

select
  o_year,
  sum(case
    when nation = 'BRAZIL' then volume
    else 0
  end) / sum(volume) as mkt_share
from
  (
    select
      year(o_orderdate) as o_year,
      l_extendedprice * (1 - l_discount) as volume,
      n2.n_name as nation
    from
      part,
      supplier,
      lineitem,
      orders,
      customer,
      nation n1,
      nation n2,
      region
    where
      p_partkey = l_partkey
      and s_suppkey = l_suppkey
      and l_orderkey = o_orderkey
      and o_custkey = c_custkey
      and c_nationkey = n1.n_nationkey
      and n1.n_regionkey = r_regionkey
      and r_name = 'AMERICA'
      and s_nationkey = n2.n_nationkey
      and o_orderdate between date('1995-01-01') and date('1996-12-31')
      and p_type = 'ECONOMY ANODIZED STEEL'
  ) as all_nations
group by
  o_year
order by
  o_year;

```

Figure 54: *q8*

```

select
  nation,
  o_year,
  sum(amount) as sum_profit
from
  (
    select
      n_name as nation,
      year(o_orderdate) as o_year,
      l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
    from
      part,
      supplier,
      lineitem,
      partsupp,
      orders,
      nation
    where
      s_suppkey = l_suppkey
      and ps_suppkey = l_suppkey
      and ps_partkey = l_partkey
      and p_partkey = l_partkey
      and o_orderkey = l_orderkey
      and s_nationkey = n_nationkey
      and p_name like '%green%'
  ) as profit
group by
  nation,
  o_year
order by
  nation,
  o_year desc;

```

Figure 55: *q9*

```

select
  c_custkey,
  c_name,
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  c_acctbal,
  n_name,
  c_address,
  c_phone,
  c_comment
from
  customer,
  orders,
  lineitem,
  nation
where
  c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate >= date('1993-10-01')
  and o_orderdate < date('1994-01-01')
  and l_returnflag = 'R'
  and c_nationkey = n_nationkey
group by
  c_custkey,
  c_name,
  c_acctbal,
  c_phone,
  n_name,
  c_address,
  c_comment
order by
  revenue desc;

```

Figure 56: *q10*

```

select
  ps_partkey,
  sum(ps_supplycost * ps_availqty) as value
from
  partsupp,
  supplier,
  nation
where
  ps_suppkey = s_suppkey
  and s_nationkey = n_nationkey
  and n_name = 'GERMANY'
group by
  ps_partkey having
    sum(ps_supplycost * ps_availqty) > (
      select
        sum(ps_supplycost * ps_availqty) * 0.0001000000
      from
        partsupp,
        supplier,
        nation
      where
        ps_suppkey = s_suppkey
        and s_nationkey = n_nationkey
        and n_name = 'GERMANY'
    )
order by
  value desc;

```

Figure 57: *q11*

```

select
  l_shipmode,
  sum(case
    when o_orderpriority = '1-URGENT'
      or o_orderpriority = '2-HIGH'
    then 1
    else 0
  end) as high_line_count,
  sum(case
    when o_orderpriority <> '1-URGENT'
      and o_orderpriority <> '2-HIGH'
    then 1
    else 0
  end) as low_line_count
from
  orders,
  lineitem
where
  o_orderkey = l_orderkey
  and l_shipmode in ('MAIL', 'SHIP')
  and l_commitdate < l_receiptdate
  and l_shipdate < l_commitdate
  and l_receiptdate >= date('1994-01-01')
  and l_receiptdate < date('1995-01-01')
group by
  l_shipmode
order by
  l_shipmode;

```

Figure 58: *q12*

```

select
  c_count,
  count(*) as custdist
from
  (
    select
      c_custkey,
      count(o_orderkey) as c_count
    from
      customer, orders
    where
      c_custkey = o_custkey
      and o_comment not like '%special%requests%'
    group by
      c_custkey
  )
group by
  c_count
order by
  custdist desc,
  c_count desc;

```

Figure 59: *q13*

```

select
  100.00 * sum(case
    when p_type like 'PROMO%'
      then l_extendedprice * (1 - l_discount)
    else 0
  end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue
from
  lineitem,
  part
where
  l_partkey = p_partkey
  and l_shipdate >= date('1995-09-01')
  and l_shipdate < date('1995-10-01');

```

Figure 60: *q14*

```
select
    s_suppkey,
    s_name,
    s_address,
    s_phone,
    total_revenue
from
    supplier,
    revenue0
where
    s_suppkey = supplier_no
    and total_revenue = (
        select
            max(total_revenue)
        from
            revenue0
    )
order by
    s_suppkey;
```

Figure 61: *q15*

```

select
  p_brand,
  p_type,
  p_size,
  count(ps_suppkey) as supplier_cnt
from
  partsupp,
  part
where
  p_partkey = ps_partkey
  and p_brand <> 'Brand#45'
  and p_type not like 'MEDIUM POLISHED%'
  and p_size in (49, 14, 23, 45, 19, 3, 36, 9)
  and ps_suppkey not in (
    select
      s_suppkey
    from
      supplier
    where
      s_comment like '%Customer%Complaints%'
  )
group by
  p_brand,
  p_type,
  p_size
order by
  supplier_cnt desc,
  p_brand,
  p_type,
  p_size;

```

Figure 62: *q16*

```
select
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    lineitem,
    part
where
    p_partkey = l_partkey
    and p_brand = 'Brand#23'
    and p_container = 'MED BOX'
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            lineitem
        where
            l_partkey = p_partkey
    );
```

Figure 63: *q17*

```

select
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice,
    sum(l_quantity)
from
    customer,
    orders,
    lineitem
where
    o_orderkey in (
        select
            l_orderkey
        from
            lineitem
        group by
            l_orderkey having
                sum(l_quantity) > 300
    )
    and c_custkey = o_custkey
    and o_orderkey = l_orderkey
group by
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice
order by
    o_totalprice desc,
    o_orderdate;

```

Figure 64: *q18*

```

select
    sum(l_extendedprice* (1 - l_discount)) as revenue
from
    lineitem,
    part
where
    (
        p_partkey = l_partkey
        and p_brand = 'Brand#12'
        and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
        and l_quantity >= 1 and l_quantity <= 1 + 10
        and p_size between 1 and 5
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    )
    or
    (
        p_partkey = l_partkey
        and p_brand = 'Brand#23'
        and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
        and l_quantity >= 10 and l_quantity <= 10 + 10
        and p_size between 1 and 10
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    )
    or
    (
        p_partkey = l_partkey
        and p_brand = 'Brand#34'
        and p_container in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
        and l_quantity >= 20 and l_quantity <= 20 + 10
        and p_size between 1 and 15
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    );

```

Figure 65: *q19*

```

select
    s_name,
    s_address
from
    supplier,
    nation
where
    s_suppkey in (
        select
            ps_suppkey
        from
            partsupp
        where
            ps_partkey in (
                select
                    p_partkey
                from
                    part
                where
                    p_name like 'forest%'
            )
        and ps_availqty > (
            select
                0.5 * sum(l_quantity)
            from
                lineitem
            where
                l_partkey = ps_partkey
                and l_suppkey = ps_suppkey
                and l_shipdate >= date('1994-01-01')
                and l_shipdate < date('1995-01-01')
        )
    )
    and s_nationkey = n_nationkey
    and n_name = 'CANADA'
order by
    s_name;

```

Figure 66: *q20*

```

select
    s_name,
    count(*) as numwait
from
    supplier,
    lineitem l1,
    orders,
    nation
where
    s_suppkey = l1.l_suppkey
    and o_orderkey = l1.l_orderkey
    and o_orderstatus = 'F'
    and l1.l_receiptdate > l1.l_commitdate
    and exists (
        select
            *
        from
            lineitem l2
        where
            l2.l_orderkey = l1.l_orderkey
            and l2.l_suppkey <> l1.l_suppkey
    )
    and not exists (
        select
            *
        from
            lineitem l3
        where
            l3.l_orderkey = l1.l_orderkey
            and l3.l_suppkey <> l1.l_suppkey
            and l3.l_receiptdate > l3.l_commitdate
    )
    and s_nationkey = n_nationkey
    and n_name = 'SAUDI ARABIA'
group by
    s_name
order by
    numwait desc,
    s_name;

```

Figure 67: *q21*

```

select
  cntrycode,
  count(*) as numcust,
  sum(c_acctbal) as totacctbal
from
  (
    select
      substr(c_phone,1,2) as cntrycode,
      c_acctbal
    from
      customer
    where
      c_acctbal > (
        select
          avg(c_acctbal)
        from
          customer
        where
          c_acctbal > 0.00
          and substr(c_phone,1,2) in
            ('13', '31', '23', '29', '30', '18', '17')
      )
    and not exists (
      select
        *
      from
        orders
      where
        o_custkey = c_custkey
    )
    and substr(c_phone,1,2) in
      ('13', '31', '23', '29', '30', '18', '17')
  ) as custsale
group by
  cntrycode
order by
  cntrycode;

```

Figure 68: *q22*

## References

- [1] MonetDB - Query Processing at Light Speed.  
<http://monetdb.cwi.nl/>.
- [2] The Embarcadero SQL Debugger.  
<http://www.embarcadero.com/products/debugger>.
- [3] Transact-SQL Debugger in Microsoft SQL Server 2008.  
<http://msdn.microsoft.com/en-us/library/cc646024.aspx>.
- [4] Transaction Processing Performance Council. TPC-H, an ad-hoc, decision support benchmark.  
<http://www.tpc.org/tpch/>.
- [5] A. Chapman and H.V. Jagadish. Why Not? *In Proc. SIGMOD*, 2009.
- [6] Fabian Kliebhan. A Truly Compositional SQL Compiler. 2009.
- [7] Hans-Joachim Ruscheweyh. A new compositional approach to SQL compilation. 2009.
- [8] Torsten Grust, Jan Rittinger. Observing SQL Queries in their Natural Habitat. 2010.