A Truly Compositional SQL Compiler

Fabian Kliebhan

Aufgabensteller: Prof. Dr. Torsten Grust
Betreuer: Jan Rittinger, M.Sc.

September 4, 2009

Summary

This thesis deals with the compilation of SQL into the Algebra of *Pathfinder* and it's implementation in Java. One intention is to create the basis to build a SQL debugger, which becomes possible whilst SQL Statements are parsed into another language. The concept and details of the implementation and the ideas behind it are topic of this thesis.

Contents

1 Introduction **5**
- 1.1 Motivation . . . 5
- 1.2 The concept . . . 5
- 1.3 Implementation . . . 5

2 ZQL **7**
- 2.1 What is *ZQL* . . . 7
- 2.2 Example . . . 7
- 2.3 *ZQL* Class representation . . . 9
- 2.4 Utilization of *ZQL* . . . 10

3 SQL AST **11**
- 3.1 Concept . . . 11
- 3.2 Compilation of *ZQL* into the AST representation . . . 11
- 3.3 The AST classes . . . 12
  - 3.3.1 The **SFW** class . . . 12
  - 3.3.2 The **FromBlock** class . . . 12
  - 3.3.3 The **WhereBlock** class . . . 12
  - 3.3.4 The **SelectBlock** class . . . 13
  - 3.3.5 The **Predicate** class . . . 13
  - 3.3.6 The **Table** class . . . 13
  - 3.3.7 The **Constant** class . . . 13
  - 3.3.8 The **Name** class . . . 13
  - 3.3.9 **All**,**Any**,**Exists** and **In** classes . . . 13
- 3.4 Example . . . 14
- 3.5 AST Class hierarchy . . . 16
- 3.6 Comparison to *ZQL* . . . 16

4 Class representation of the *Pathfinder* Algebra **17**
- 4.1 Concept . . . 17
- 4.2 Example . . . 18
- 4.3 Integration in the compilation process . . . 19

5 Implementation of compilation Rules **20**
- 5.1 Concept . . . 20
- 5.2 The Rules . . . 21
- 5.3 Example . . . 21

6 Putting it all together **23**
- 6.1 Concept . . . 23
- 6.2 Example . . . 23

7 Recapitulation **26**

Index **27**

References **28**

1 Introduction

1.1 Motivation

The main idea behind converting SQL to the *Pathfinder* Algebra is the possibility to convert SQL queries into a lot of different backend languages. The other opportunity that arises is that with the resulting Java structures, that are built to make the compilation, it is possible to set the basic requirements to build a debugger for SQL. The Java class structure that is created for the *Pathfinder* Algebra could also be useful in other work, as its modular and flexible representation can be used in a lot of different ways. One example is a GUI tool written in Java to create *Pathfinder* queries. Also this class representation can be used for compilation of other languages to the *Pathfinder* Algebra.

1.2 The concept

The underlying concept is to convert SQL statements into relational algebra plans of the *Pathfinder* Algebra (represented in XML). Here, we implement rules devised by Hans-Joachim Ruscheweyh [5] in Java to make this compilation possible. These Rules will be applied on input SQL queries and they will produce *Pathfinder* XML code as output. These queries can be optimized very easily in *Pathfinder*. Also *Pathfinder* observations offer the possibility to take a look at partial result as well as the structure of the evaluation of the code.

1.3 Implementation

This thesis is about the implementation of rules to convert SQL into the *Pathfinder* Algebra. The programming language that is used to achieve this goal is Java.
The first task is to convert SQL statements into a class representation, that can be handled by Java. *ZQL* is an open source library that is used to convert simple SQL statements into this structure. For more complex routines in SQL a pre-parser will establish an accurate representation in *ZQL*. The *ZQL* part is covered in Chapter 2. After that an abstract syntax tree (AST) is created for the input SQL query. The main purpose of the AST is to have a more modular access to the SQL query. Chapter 3 covers the AST in more detail. These AST classes serve as input for the final compilation code that is based on the compilation Rules written by Hans-Joachim Ruscheweyh[5]. By applying these Rules, the SQL query is converted to instances of a Java class representation of the *Pathfinder* Algebra operators, which is built with the underlying concept of the *Composite pattern* and is therefore also (like the SQL AST) a tree structure. The creation of this class representation which represents the *Pathfinder* operators is also a component of this thesis and is covered in Chapter 4. The concrete implementation of the compilation Rules is

part of Chapter 5. The different steps that need to be taken are shown in Figure 1.

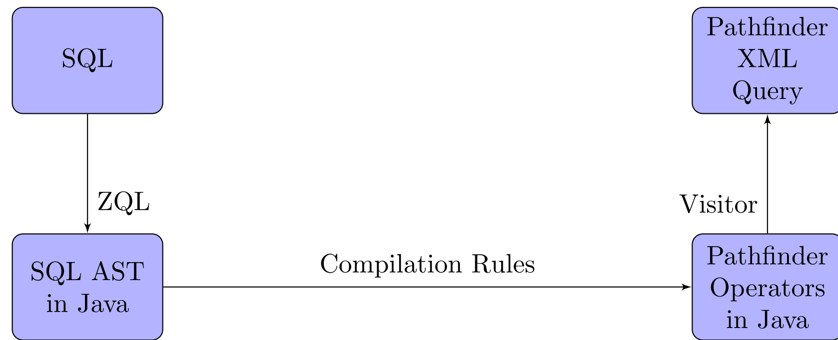

Figure 1: Compilation process

2 ZQL

2.1 What is *ZQL*

ZQL is an open source Java class library, that offers the possibility to convert SQL Statements into a Java class representation. The starting point to do this is the **ZqlParser** class. Passing an **InputStream** to the constructor of this class lets you create an instance on which you can call the **parse()**-function that starts reading statements of this Input Stream. For example one can instantiate an Object with **System.in** as argument for the constructor and use the standard input to receive queries. Another way would be to pass a **ByteArrayInputStream** Object to the constructor and parse simple character String queries (see **2.2**). The returning Object of the **parse()** function has the type **ZStatement** that is (if the query was valid) a super type of the **ZQuery** object. On this object you can call various functions like **getFrom()**, **getSelect()** or **getWhere()**. Those functions will return objects of a few more *ZQL* types like **ZFromItem**, **ZSelectItem** or **ZExpression**. This continues in a recursive way and covers the whole SQL query.

2.2 Example

Program 1 shows Java code that parses the SQL Query *Q1* into the *ZQL* class representation and prints the From, Where and Select items. Figure 2 shows the class tree of the resulting objects, that are created. One can see that the query is parsed into a tree structure where every subpart is represented by an appropriate Java class. We have different Objects for the Select, From and Where parts (**ZSelectItem**, **ZFromItem**, and **ZConstant**).

Query *Q1*:

```
SELECT  ID,COUNTER
FROM    SALES
WHERE   COUNTER > 10;
```

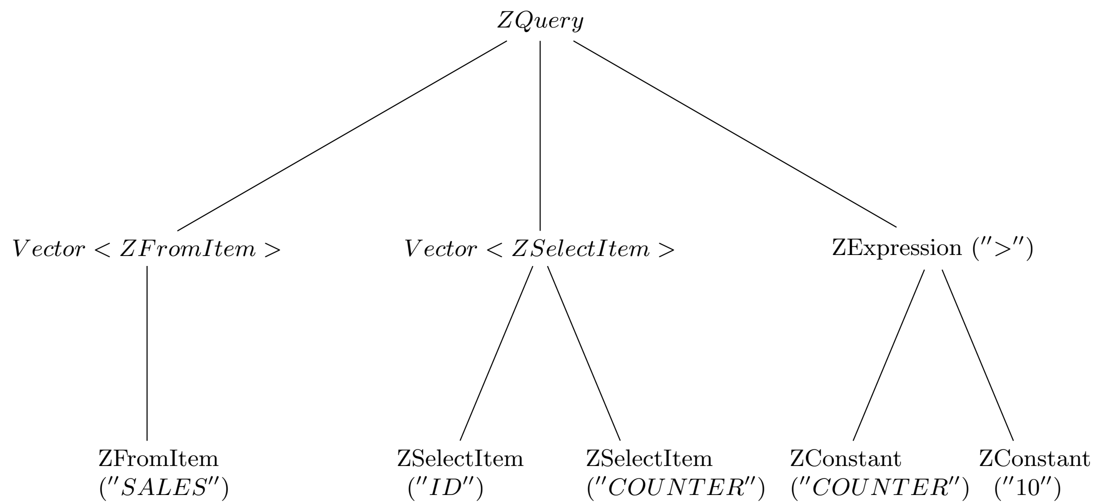

Figure 2: *ZQL* class representation of Query *Q1*

```
1  String sqlquery = "SELECT ID,COUNTER FROM SALES WHERE COUNTER > 10;";
2  //create stream for the ZQLParser
3  ByteArrayInputStream s = new ByteArrayInputStream(sqlquery.getBytes());
4  // creates the parser
5  ZqlParser parser = new ZqlParser(s);
6  // parses the query
7  ZQuery zqlquery = (ZQuery) parser.readStatement();
8  // prints from items
9  System.out.print("From Items: \t");
10 for(int i=0; i<zqlquery.getFrom().size();i++)
11    System.out.print(zqlquery.getFrom().get(i)+ " ");
12 // prints select items
13 System.out.print("\nSelect Items: \t");
14 for(int i=0; i<zqlquery.getSelect().size();i++)
15    System.out.print(zqlquery.getSelect().get(i)+ " ");
16 // prints where item
17 System.out.println("\nWhere Item: \t"+zqlquery.getWhere().toString());
```

Program 1: Parses a SQL query into the *ZQL* representation

Output:

```
From Items:      SALES
Select Items:    ID COUNTER
Where Item:      (SALES.COUNTER > 10)
```

2.3 *ZQL* Class representation

Table 1 shows the class representation of the most important SQL parts in *ZQL*. Be aware that the list is not complete. *ZQL* offers many more Objects. For a more detailed view you can take a look at the Java documentation of *ZQL* at [3].

SQL Objects	**ZQL Representation**
SQL query	ZQuery
From Block	Vector<ZFromItem>
From Item	ZFromItem
Select Block	Vector<ZSelectItem>
Select Item	ZSelectItem
Where Expression	ZExpression
Subexpression	ZExpression
Constant (like 4,2.1,”string”,...)	ZConstant
Column	ZConstant
Table	String
Alias	String
Schema	String
Group By	ZGroupBy
Order By	ZOrderBy
Update	ZUpdate
Delete	ZDelete
Insert	ZInsert

Table 1: *ZQL* Class representation

2.4 Utilization of *ZQL*

ZQL is used to convert the input SQL queries in a Java class structure. The main SFW block will be parsed into **ZQuery** object, from which it is possible to gain access to the underlying sub objects. These *ZQL* structures build the basis to create a SQL abstract syntax tree, that is needed for a SQL debugger and is also used to start the compilation process without directly accessing *ZQL* Objects. This results in a more modular and flexible designed overall project. Figure 3 shows the context where *ZQL* is used in the compilation process.

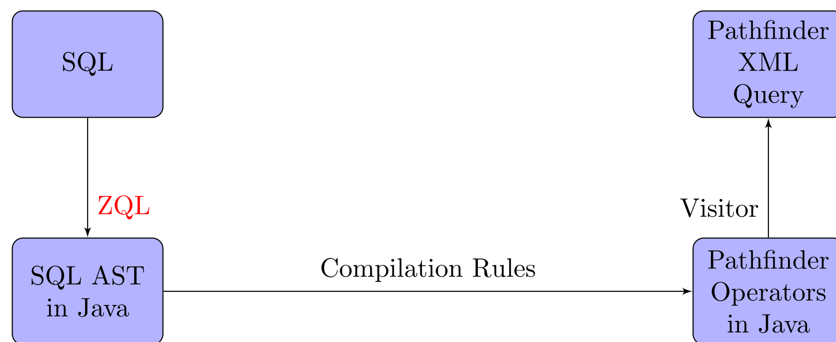

Figure 3: Role of *ZQL* in the compilation process

3 SQL AST

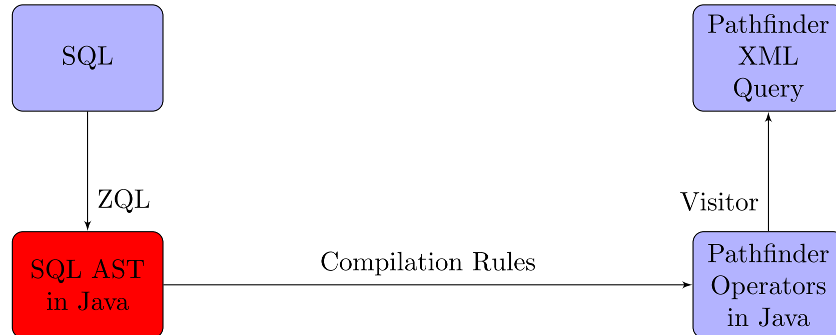

Figure 4: SQL AST in the compilation process

3.1 Concept

The next thing to do is building up a SQL abstract syntax tree, from where the compilation is able to start. The AST is used for modular access to the SQL query. The idea is that the *ZQL* backend can be replaced without changing the compilation Rules. The only thing that needs to be changed is the ASTParser, which parses the AST (here: ZQL2ASTParser).
The AST contains classes which encapsulate the different structures that appear in a SQL query. Also these classes are directly used by the compilation process, that is the main topic of this thesis. The exact class tree is shown in Figure 5. Every class in the AST extends the **SQLAST** superclass.

3.2 Compilation of *ZQL* into the AST representation

The parsing of the *ZQL* Objects into the AST representation is done by the **ZQL2ASTParser** class. This process is done in different steps. First of all the **ZQuery** Object is passed to the **ZQL2ASTParser** constructor. After that the From, Where and Select parts are parsed one by one. In the last step all subexpressions inside the Select, From or Where parts are recursively parsed. The result is a **SFW** Object which contains the complete SQL AST.

3.3 The AST classes

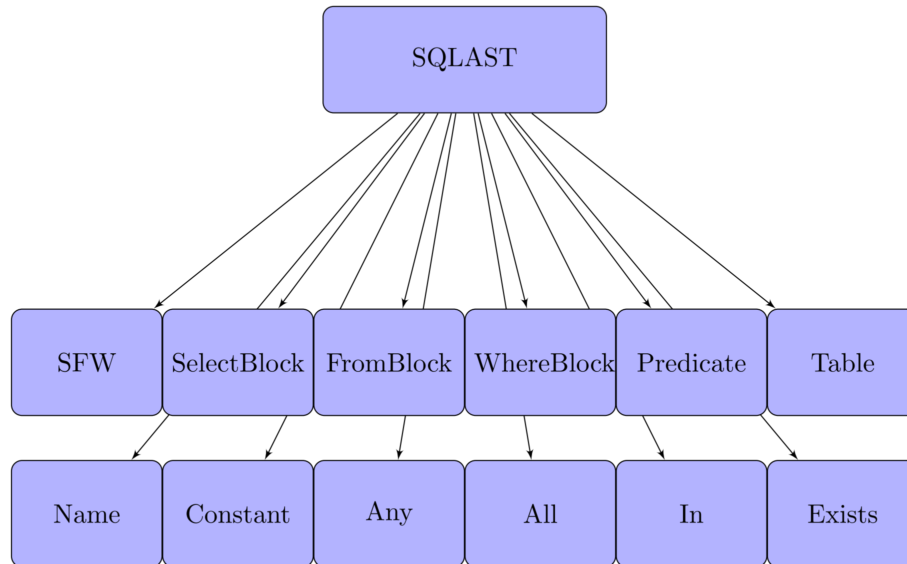

Figure 5: AST class hierarchy

3.3.1 The SFW class

The main structure in the AST is the SWF block that is represented by a Java class called **SFW**. This class contains references to its substructures. These classes are the **FromBlock** class, the **WhereBlock** class and the **SelectBlock** class. Also it contains the alias name for the current query if one is given.

3.3.2 The FromBlock class

The **FromBlock** class contains an array (**Vector** Object) of **SQLAST** Objects. These **SQLAST** Objects are representing the elements in the FROM clause of the SQL statement. These Objects can be instances of different AST classes like the **SFW** or the **Table** class.

3.3.3 The WhereBlock class

The **WhereBlock** class contains only one **SQLAST** Object. This **SQLAST** Object can be of different types depending of the concrete expression. For example if the where block starts with an EXISTS expression, the SQLAST Object is an instance of the **EXISTS** class.

3.3.4 The SelectBlock class

The **SelectBlock** class contains **SQLAST** Objects that represent the columns referenced in the Select block. These **SQLAST** Objects are instantiated, again recursively, in the constructor.

3.3.5 The Predicate class

The **Predicate** class contains references to the underlying **SQLAST** Objects and the operator type. The expression 'COUNTER < 5' e.g., stores the Operator type '<' and references a **Name** ('COUNTER') and **Constant** ('5') instance.

3.3.6 The Table class

Objects of this class are referenced (and created) by the **FromBlock** class. They are representing SQL tables and contain the table names and aliases.

3.3.7 The Constant class

This class represents Number or String constants. The type and value of a constant are stored here.

3.3.8 The Name class

The **Name** class contains column names. An instance represents a column that is referenced inside a query. If the same column is referenced twice inside a query, two different **Name** instances with the same column name are created.

3.3.9 All,Any,Exists and In classes

These classes are representing the corresponding operators. They contain the referenced subexpression and, if necessary the operator type (<, >, <=, >=, ...).

3.4 Example

Program 2 shows how the *ZQL* representation of Query *Q1*, created by Program 1, is converted into the AST representation.
The AST representation differs from the *ZQL* representation in a few points. There are for example no different classes for Select or From Items. Therefore the representation is more dynamic and modular as these classes are placed depending on their type, unlike in *ZQL* where the class type depends on the location.

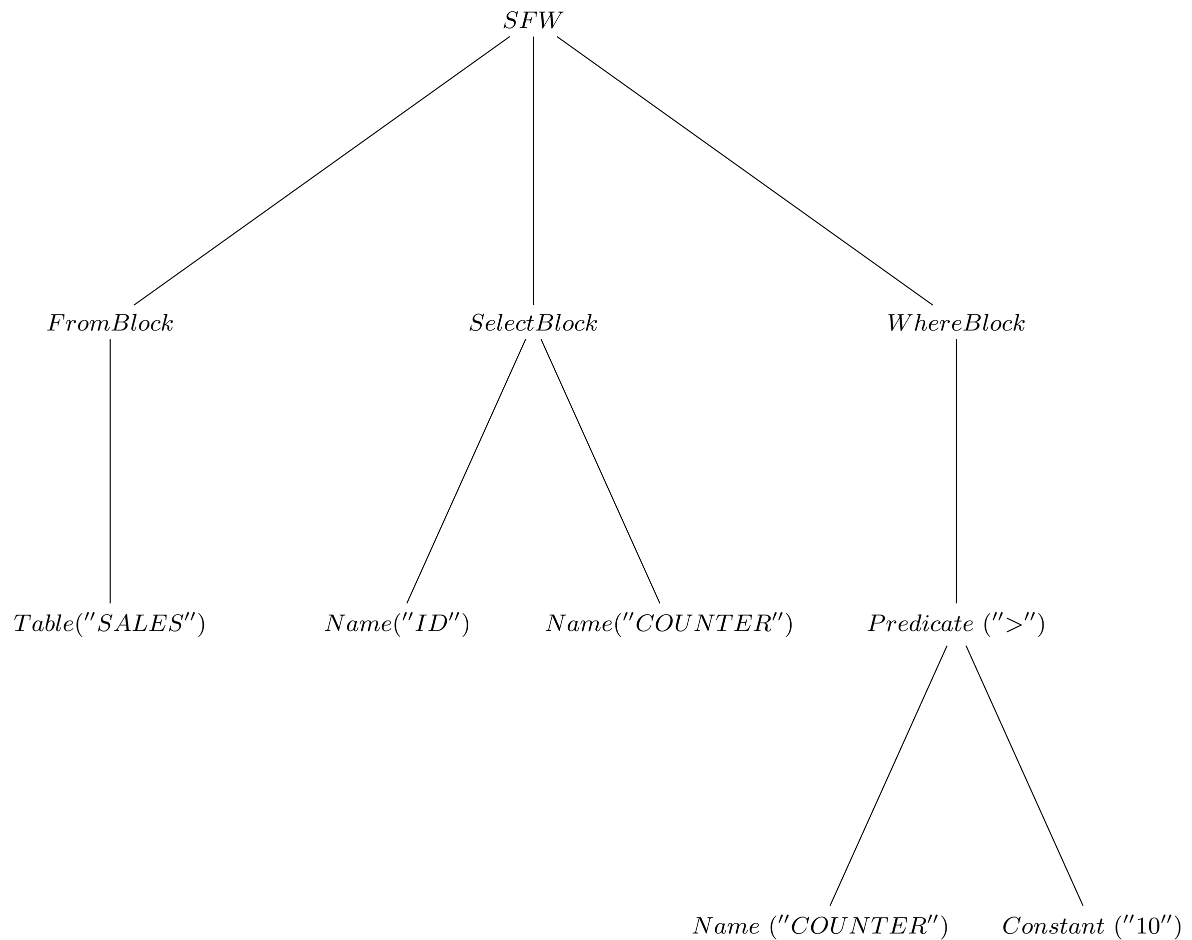

Figure 6: SQL AST class representation of Query *Q1*

```
1 ZQL2ASTParser zql2ast = new ZQL2ASTParser(new SQLQuery(zqlquery));
2 SFW sfw = zql2ast.parseAST();
3 System.out.print("From Items: \t");
4 for(int i=0; i<sfw.getFrom().getFroms().size();i++)
5     System.out.print(sfw.getFrom().getFroms().get(i)+ " ");
6 System.out.print("\nSelect Items: \t");
7 for(int i=0; i<sfw.getSelect().size();i++)
8     System.out.print(sfw.getSelect().getSelectItems()[i]+ " ");
9 Predicate predicate = (Predicate) sfw.getWhere().getExp();
10 System.out.print("\nWhere Item: \t" + predicate.getExprs()[0]
11                                     + predicate.getOperator()
12                                     + predicate.getExprs()[1]);
```

Program 2: Parses the *ZQL* representation into the AST representation

Output:

```
From Items:        SALES
Select Items:      ID COUNTER
Where Item:        COUNTER>10
```

3.5 AST Class hierarchy

Table 2 shows the class representation of the AST. Note that the AST is not completed yet. Operations like 'group by' or 'order by' are not supported for now.

SQL Objects	**AST Representation**
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
Alias	String
Schema	String
Group By	n.a.
Order By	n.a.
Update	n.a.
Delete	n.a.
Insert	n.a.

Table 2: SQL AST Class hierarchy

3.6 Comparison to *ZQL*

The main difference to the *ZQL* class representation is that the SQL AST has no separate classes for the Select items, From items, Where expressions and so on. Instead the referenced Objects like **Name**, **Constant** or **Table** are stored directly. This means we don't need strict separation between the Select, From or Where items. Instead they are created dynamically depending on the referenced Object in the query. By using only the needed references and not wrapping them according to their appearance the design becomes more compact, flexible and modular.

4 Class representation of the *Pathfinder* Algebra

4.1 Concept

A class representation of the *Pathfinder* Algebra is necessary to accomplish the compilation of the SQL AST objects into the *Pathfinder* Algebra. It is an intermediate stage that is necessary to simplify and unitize the compilation. To accomplish this, various *Pathfinder* Operators are brought to a Java class representation that stores information of the various Operators like kind, referenced tables and columns, child Operators, new columns and so on. These operator Objects can not execute the Algebraic operations by themselves but the whole query (represented with the root Object **Serialize_Rel**) can be converted to XML that can be parsed by the *Pathfinder* system. This is done using the *Visitor*-pattern.
Figure 7 shows parts of the class hierarchy of the *Pathfinder* Operators. Information about all classes can be found here [4]. The common superclass of all Operators is the **Operator**-class, which contains basic properties of all Operators (among other things kind and id of the operator). Furthermore it contains a accept and reset function for the visitor. 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 are representing the original *Pathfinder* Operators. These Operators can be instantiated by passing all information that is needed by the particular Operator to the constructor. Furthermore you need to pass references to the child Operators on non leaf nodes. Once all Operators of a query are created you can call the **DAGaccept()** function on the root Operator **SerializeRel** Object by passing an instance of the **XMLCreator** visitor class that will create the XML output. Now the **toString()** function of the **XMLCreator** provides the XML code that can be written to a file and be used as Input for *Pathfinder*.

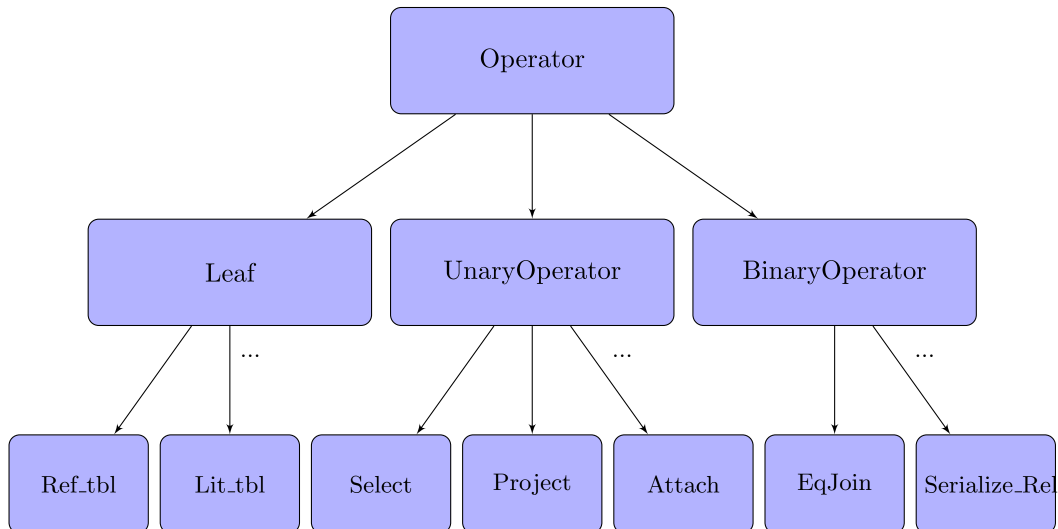

Figure 7: *Pathfinder* Operators class hierarchy

4.2 Example

Program 3 shows how the *Pathfinder* representation of the example SQL Query *Q1* can be built using the *Pathfinder* Java Operators. Note that normally this is done in a dynamic way by the compilation Rules covered in Chapter 5. This example just shows how it can be done using the Java class representation of the *Pathfinder* Operators.

```
1 Ref_tbl table = new Ref_tbl("SALES",
2                 new String[]{"item1"        ,"item2"},
3                 new String[]{"ID"    ,"COUNTER"},
4                 new String[]{"str"          ,"int"});
5 Attach att = new Attach(table,"item3","int","10");
6 LogicAndCompareOperations eq =
7     new LogicAndCompareOperations(att,"item2","item3","item4","gt");
8 Select sel = new Select(eq, "item4");
9 Project proj2 = new Project(sel, new String[]{"item1","item2"});
10 Serialize_Rel ser =
11     new Serialize_Rel(
12                 new NIL(),proj2,"item1","item2",
13                 new String[]{"item1","item2"});
14 XMLCreator xmlcreator = new XMLCreator();
15 xmlcreator.visit(ser);
16 IO.writeToFile(xmlcreator.toString(),outputfilename);
```

Program 3: Creates the *Pathfinder* representation of Query *Q1*

As mentioned before the Operator classes used in this examples are wrappers for the operators of the *Pathfinder* Algebra.
The code creates a query that, if executed, first creates a table, that references the SQL table 'SALES'. The columns of the table are renamed to *item1* and *item2*. Then a new column (*item3*) with "10" values is attached. After that the column *item2* and *item3* are compared by 'greater than' and a new column (*item4*) with 'true' and 'false' values is created. The Select Operator eliminates all rows with 'false' value on column *item4* and the Project Operator eliminates all columns but *item1* and *item2*. Last but not least the output is generated by the Serialize Operator.
After all Operators are instantiated the **XMLCreator** translates the query to the XML representation. Last the xml-query is stored in a file. For a more detailed view of the concrete *Pathfinder* Algebra take a look [2]. More information about the XML output can be found at [1].

4.3 Integration in the compilation process

In contrary to the SQL AST which represents the SQL queries, the Operator classes are representing the *Pathfinder* Operators. Instances of these Objects will be created while the compilation process takes place. Following the Rules created by Hans-Joachim Ruscheweyh [5] a whole Pathfinder query will be instantiated step by step based on the SQL AST. Figure 8 shows the context where this compilation takes place.

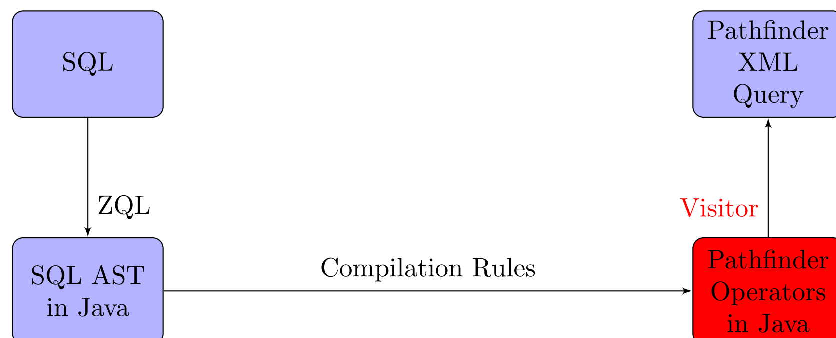

Figure 8: *Pathfinder* Operators in the compilation process

5 Implementation of compilation Rules

5.1 Concept

The compilation Rules are divided in different functions (which are coupled to classes). For every Rule in Hans-Joachim Ruscheweyh's paper [5] a corresponding Java class does exist. A Rule is started by calling the **compile()** function of the depending **Rules** class. In Hans-Joachim Ruscheweyh's work there are two types of interference Rules. One type returns a column and the other type returns a table (see also [5, Section 3.2]). Here the two functions **compileTable()** and **compileColumn()** represent the inference process. In contrast to the initial **compile()** function the **compileTable()** and **compileColumn()** functions are generic functions that are located in the **Rules** class. They take a **SQLAST** instance as argument and call the appropriate Rule depending on the subtype of this instance. Therefore the SQL AST input defines which Rule will be started. For example if the compilation input contains an **SFW** Object (defined in the SQL AST), the SFW Rule will be called by the generic **compileTable()** function. This generic function, in turn, is called by the different Rules, except for the first time, where the static **compile()** function of the **Rules** class starts the compile process. This process continues with every **SQLAST** Object that is present. That means for every **SQLAST** Object the corresponding Rule is started until the whole query is parsed. The **Rules** class is the superclass of all particular Rules like the already mentioned **SFWRule**, the **FromRule** Object, the **WhereRule** Object and so on. Figure 9 shows the actual context in the overall compilation process.

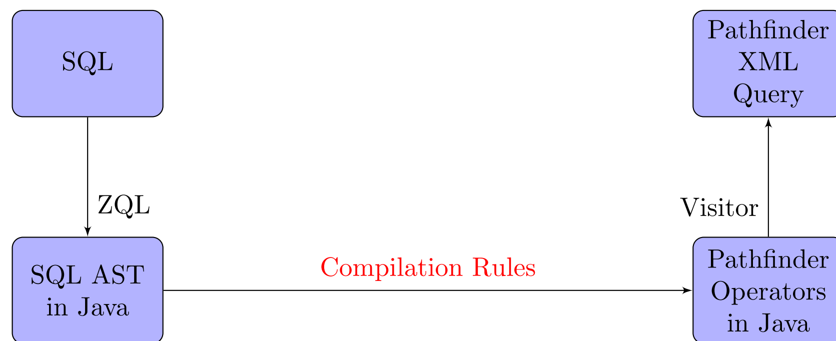

Figure 9: The Rules in the compilation process

5.2 The Rules

The compilation process starts by calling the **compile()** function of the **Rules** class. This functions takes a **SFW** Object as argument which is part of the SQL AST and returns a **Serialize_Rel** Object that represents the *Pathfinder* Algebra Output. It creates the starting Loop and Environment (Objects, that are used for the compilation) and then starts the Serialize Rule. This is the first time where the Rules of Hans-Joachim Ruscheweyh [5] come to play. The following Rules are started by the generic **compileTable()** and **compileColumn()** functions depending on the given SQL AST arguments. These Rules then continue this process by calling the generic functions until the whole query is compiled.
If we take a look at the example Query *Q1* the first Rule that is started is the Serialize Rule (as described before). This Rule now calls the generic **compileTable()** function with the **SFW** Object of Query *Q1* as argument.
The generic function starts the SFW Rule caused by the given **SFW** argument. The Rules that are triggered by the SFW Rule are the From Rule, followed by the Where Rule and at last the Select Rule.
The From Rule again triggers the Table Rule (for the 'SALES' table), the Where Rule triggers the Predicate Rule (for the 'COUNTER > 10' expression) and the Select Rule triggers two times the Name Rule (for the columns 'ID' and 'COUNTER').
The Predicate Rule starts the Name and Constant Rules for the column 'COUNTER' and constant '10'.
For a more detailed view of the compilation process take a look at [4] and [5].

5.3 Example

Equation 1 shows Hans-Joachim Ruscheweyh's representation and Program 4 the Java code of the Exists Rule. This will illustrate how the Rules are represented in Java. The **compile()** function takes (like every Rule function) an Environment and Loop as argument. The third argument is the AST object. The function returns an **Operator** that represents a column (the other possibility would be returning a new Environment and Loop, which would represent a table). The compilation itself is done exactly like the Rules specify it. The example shows how the Java equivalents to the attach, project and disunion operators are used.

$$\Gamma; loop \vdash e \Mapsto (\Gamma'; loop') \quad (1)$$

$$q \equiv \delta(\pi_{(\mathsf{KEY:OK})} loop') \quad (2)$$

$$q' \equiv (@_{(\mathsf{ITEM}='true')} q) \cup (@_{(\mathsf{ITEM}='false')} (\pi_{(\mathsf{KEY:IK})} (loop)) \backslash q) \quad (3)$$

$$\overline{\Gamma; loop \vdash \texttt{EXISTS}\ e \rhd q'}$$

Equation 1: Exists Rule

```
1  public static Operator compile(Environment oldEnvironment,
2          Operator oldLoop, WhereBlock whereBlock) {
3      SQLAST e = exists.getExistsExpr(); // (1)
4      Operator loop_ = compileTable(oldEnvironment, oldLoop, e).loop; // (2)
5      Operator q = new Distinct(
6                      new Project(
7                          loop_,
8                          new String[]{key()},
9                          new String[]{ok()})); // (3)
10     Operator q_ = new Disunion(
11                     new Attach(
12                             q,
13                             item(),
14                             "bool",
15                             "true"),
16                     new Attach(
17                             new Difference(
18                                     new Project(oldLoop,
19                                             new String[]{key()},
20                                             new String[]{ik()}),
21                                     q),
22                             item(),
23                             "bool",
24                             "false"));
25     return q_;
26 }
```

Program 4: Java Code for the Exists Rule

6 Putting it all together

6.1 Concept

Now that all parts of the process are defined, they have to be put together. Program 5 shows how the different aspects communicate with each other. It is at large the same code that is placed in the **main()** function to run the compilation.

```
1 /*
2 step 1. Reading SQL input
3 */
4 String sqlInput = IO.readFromFile(inputFileName);
5 /*
6 step 2. Parsing ZQL
7 */
8 SQLQuery zqlQuery = new SQLPreParser().parse(sqlInput);
9  /*
10 step 3. Parsing AST
11 */
12 ZQL2ASTParser astParser = new ZQL2ASTParser(zqlQuery);
13 SQLAST ast = astParser.parseAST();
14 /*
15 step 4. Compiling Pathfinder Operators
16 */
17 Serialize_Rel relAlg = Rules.parse(ast);
18 /*
19 step 5. creating XML Output
20 */
21 XMLCreator xml = new XMLCreator();
22 relAlg.DAGaccept(xml);
23 String xmlOutput = xml.toString();
24 /*
25 step 6. Writing XML Output
26 */
27 IO.writeToFile(xmlOutput, xmlOutputFileName);
```

Program 5: Putting it all together

6.2 Example

Figure 10 shows the resulting *Pathfinder* Algebra plan that will be created after going through all the compilation steps, using again the example Query *Q1*. Figure 11 shows the same query after it is optimized by *Pathfinder*.

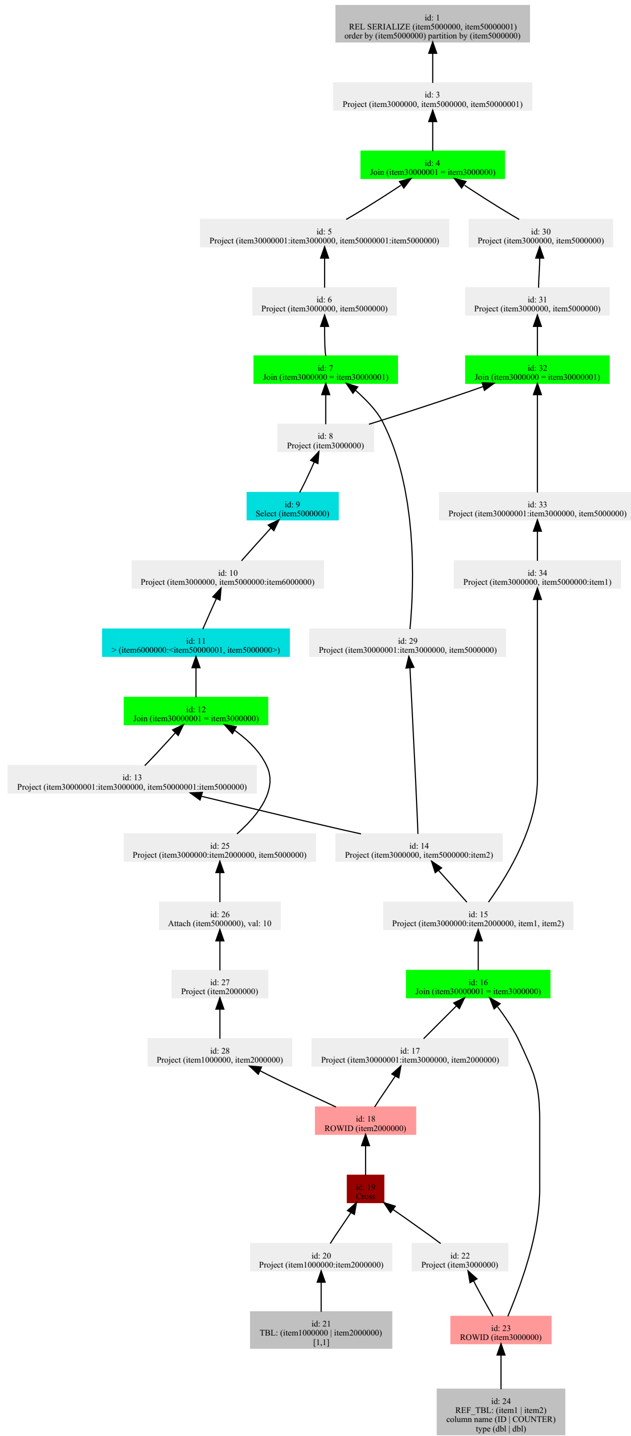

Figure 10: Unoptimized *Pathfinder* Algebra Plan of Query *Q1*

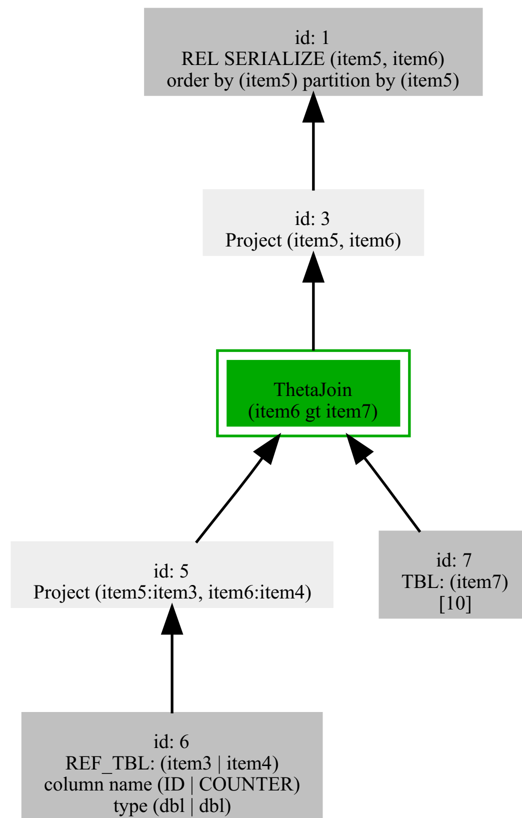

Figure 11: Optimized*Pathfinder* Algebra Plan of Query *Q1*

7 Recapitulation

This thesis started with a clear concept. Its goal was to find a modular way to implement the compilation Rules of Hans-Joachim Ruscheweyh [5]. This was accomplished by first building a Java class structure that represents a SQL input query. With help of the open source library *ZQL* the foundation for the concrete representation, which is done by the AST, was built. The AST extends the possibilities of *ZQL* with a more flexible structure and results in a more modular overall structure. The other thing that was needed was a representation of the resulting *Pathfinder* Operators. These Operators were built as a tree structure, where **BinaryOperator**, **UnaryOperator** and **Leaf** are superclasses of the different *Pathfinder* Operators. The compilation Rules build the different Operator instances by compiling the AST instances that represent the input query. This process is done by several functions that are encapsulated in appropriate classes and are called always by one generic function. This generic type conceals complexity by calling the appropriate function depending on the input. At last a *Visitor* class creates the *Pathfinder* XML output which is then written to a file. By using a *Visitor* the Operator classes don't need to know anything about the format of the output. This again makes the overall project more modular because one can change the output format by simply writing a new *Visitor*.
All these flexible and modular designed parts lead to a big and powerful overall structure which builds a stable and solid base for future research, like the SQL debugger. By using its extensible design, a lot of projects can build up upon this work. This eventually could lead to new technics and innovations.

Index

All, 12, 13
All class, 13
Any, 12, 13
Any class, 13
Attach, 17

BinaryOperator, 17
ByteArrayInputStream, 7

compilation Rules, 20
Constant, 12, 13
Constant class, 13

EqJoin, 17
Example Query, 7
Exists, 12, 13
Exists class, 13
Expression, 12
Expression class, 12

FromBlock, 12
FromBlock class, 12
FromRule, 20

Hans-Joachim Ruscheweyh, 5, 19–21, 26

In, 12, 13
In class, 13

Java, 5

Leaf, 17

Name, 12, 13
Name class, 13

Predicate, 12, 13
Predicate class, 13
Projection, 17
Prospect, 26

Recapitulation, 26

SelectBlock, 12, 13
SelectBlock class, 13
Serialize_Rel, 17
SerializeRel, 17
SFW, 12
SFW class, 12
SFWRule, 20
SQL debugger, 2, 5, 10, 26
Summary, 2

Table, 12, 13
Table class, 13

UnaryOperator, 17

Visitor-pattern, 17

WhereBlock, 12
WhereBlock class, 12
WhereRule, 20

XMLCreator, 17

ZExpression, 7
ZFromItem, 7
ZQL, 5, 7
ZQL2ASTParser, 11
ZQLExample, 7
ZQLParser, 7
ZQuery, 7
ZSelectItem, 7
ZStatement, 7

References

[1] Algebra XML Output. 2009.
http://wiki.pathfinder-xquery.org/wiki/index.php/Algebra_XML_Output.

[2] Logical Algebra. 2009.
http://wiki.pathfinder-xquery.org/wiki/index.php/Logical_Algebra.

[3] ZQL class documentation. 2009.
http://www.gibello.com/code/zql/api/Zql/package-tree.html.

[4] Fabian Kliebhan. Java source code of the **SQL2PF** project. 2009.

[5] Hans-Joachim Ruscheweyh. A new compositional approach to SQL compilation. 2009.