Query Processing and Optimization using Compiler Tools

Caetano Sauer
University of Kaiserslautern
Germany
csauer@cs.uni-kl.de

Karsten Schmidt
University of Kaiserslautern
Germany
kschmidt@cs.uni-kl.de

Theo Härder
University of Kaiserslautern
Germany
haerder@cs.uni-kl.de

ABSTRACT

We propose a rule-based approach for (X)Query compilation that operates on a single query representation—called abstract syntax tree (AST)—throughout the whole translation and transformation process. For this purpose, we exploit some new features of the ANTLR compiler generator such as tree pattern matching. This approach avoids error-prone transformations into different internal query representations and allows to specify grammar rules instead of writing code tailor-made for the specific DBMS. This approach may further help to develop new query optimizer rules, to prove correctness of rules, and to evaluate rule sets in a declarative manner.

1. INTRODUCTION

A query processor is part of a DBMS responsible for translating declarative statements written in a given query language into a sequence of physical operations to be executed by the lower levels of the system. This complex task includes parsing, modifying, and optimizing the initial representation of the query. Typically, this task is divided into smaller and more specific steps thereby introducing different query representations facilitating the translation step at hand. Reaching the final executable code for a given query, the query representation usually undergoes a metamorphosis from a string, via a syntax tree, to multiple equivalent logical and physical plans.

Despite the different representations, which also introduce new formalisms, data structures, and significant amounts of source code, the overall task nevertheless is (from a more abstract perspective) the translation of a declarative statement into executable code—and this problem is tackled by classical tools known to all computer scientists, namely compilers.

Technically speaking, all stages of query processing are performing essentially the same processing steps, that is, representing the query as a tree of operations and manipulating it in some specific way, which includes pattern matching, rule evaluation, and translation. To avoid writing huge amounts of sophisticated code by hand, effective language processing tools called compiler generators (or compiler-compilers) are used. In this way, programmers are enabled to create powerful and complex compilers using just a formal description of the language: *grammars*. The goal of this paper is to introduce this compiler approach in all stages of query processing, thereby achieving complexity reduction, improved maintainability, and a formal specification through grammars, which in turn provides safety.

GvD Workshop'10, 25.-28.05.2010, Bad Helmstedt, Germany

2. QUERY PROCESSING PIPELINE

The work in [4] describes all stages of XQuery[1] processing in XTC, a native XML database system [3]. XTC follows classical conventions of query processor design and will be used in this work—without loss of generality and independent of the data model—as a base reference and an example. All nine stages illustrated in Figure 1 correspond to operations over a distinct representation of a query, and the overall sequence of stages is here referred to as the *pipeline* of query processing. Three subtasks can be identified, namely translation, optimization, and execution, each embracing one or more stages. For this reason, three different internal query representations are used: AST (abstract syntax tree), XQGM (XML query graph model, i.e., logical plan), and QEP (query execution plan, i.e., physical plan). Note that the input XQuery string is considered as an external representation.

The subtask **translation** parses XQuery statements and produces ASTs, which contain the syntax elements of the query. Throughout the subsequent steps of *normalization*, *static typing*, and *simplification*, this syntax is checked and reduced to a canonical AST, which is finally translated into a logical plan described in XQGM, i.e., an XML extension of the classical query graph model (QGM)[2]. The logical plan serves as input to the **optimizer**, which reorganizes or replaces the algebraic operations. Besides reducing the number of operations and the intermediate result size, it also identifies join options and generates alternatives for logically equivalent XQGMs. The logical plan identified as optimal (i.e., cheapest) one is then converted into a physical plan, i.e., algebraic operations are replaced by physical operations on data storage objects. This physical plan, referred to as QEP, is processed by the **query engine** and delivers a sequence of XML nodes. Eventually, these nodes are *materialized* to a given result format, such as string or XML nodes.

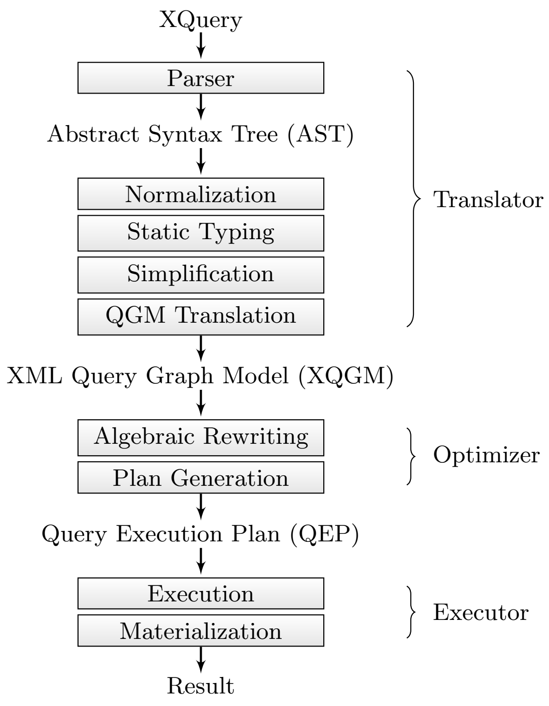


Figure 1: Query processing pipeline used in XTC

Although working at different levels of abstraction and manipulating different representations of a query, all stages of the pipeline perform essentially similar tasks, such as scanning trees (note, plans are realized as trees), identifying patterns, and performing rewrites. In the following section, we present a query processing approach that performs these general tasks based on a high-level descriptive language on a single internal representation, which is used in all stages of the pipeline.

3. ASTS AND GRAMMARS

Most of the complexity behind the implementation or maintenance of a query processor results from the use of different levels of abstraction, which are expressed by the three different internal representations in Figure 1. Implementing extensions of the query language, adding new functionalities, or even maintaining the code due to API modifications become cumbersome tasks for the programmer, particularly in research prototypes, where changes frequently happen.

3.1 Representing Queries and Plans as ASTs

A first step towards complexity reduction can certainly be achieved by replacing the three internal data structures (AST, XQGM and QEP) with only the AST. The reason for choosing a plain syntax tree for a unified query representation comes from the fact that such trees are the basic unit of work in language compilers. This chapter demonstrates techniques to execute all stages of the pipeline with ASTs while maintaining the semantics of the query language. Note that the logical abstraction of the query pipeline model is retained, while only the *implementation technique* is unified.

An AST is a labeled, ordered tree structure, where each node is a *token*. By definition, a token is the minimal syntactic unit of a language, such as a variable name or keywords like **for**, **where**, and **return**. Every token is associated to an optional string, which carries the exact sequence of characters that were matched in the input query. To illustrate the structure and the creation of the first AST, we use the sample query in Figure 2a. For this query, the parsing stage builds the AST shown in Figure 2b. Tokens are described by their type (always starting with an upper case character), e.g. **FunctionCall**, and the applicable text, which serves here as a payload and is described between brackets. In the function call token, for example, the text payload is [**"count"**]. The relationship between tokens is expressed by the tree hierarchy. The argument of the function *count*, for instance, is a variable reference represented as a child of **FunctionCall**.

3.2 Creating AST Structures from Grammars

The core engine of a query processor based on ASTs and grammars is a compiler generator. The input string is tokenized and processed by a compiler. This compiler producing the AST was generated for a specific language described by a given grammar. Such a grammar consists of a set of *rules* used to describe the elements of the language. An XQuery parser, for example, would make use of four rules to match the input of Figure 2a. At the topmost level, the query contains a FLOWR expression consisting of an assignment of a variable to a path expression and a function call with a variable reference as the return expression. These rules are described by Figure 2c, using the syntax of the ANTLR compiler generator[1].

In addition to parsing the input, the generated compiler must also execute actions embedded in the rules. One example is the maintenance of variable references in a symbol table when a variable assignment is parsed. This action appears between curly braces, like the pseudo-function $insertVariableReference()$ in the rule **flowrExpr** of Figure 2c. Later on, when a variable is referenced, the compiler looks up its value in the symbol table and replaces the reference by its value.

A special kind of actions are the so-called *rewrite rules*, which appear after the `->` symbol in the grammar. They specify how the input matched is translated to an AST, as occurring in the rule **functionCall**. Here, the tokens matched are mapped to a tree where the root node is the **FunctionCall** token and the arguments of the function are the children. The syntax used for rewrite rules is, for example, `^(a b c)`, where *a* is the root node and *b* and *c* are its children. Note, the difference between the token and the rule having apparently the same name. The token is a real object living in memory and starting with an upper-case letter. Rules, on the other hand, serve as logical units of the compiler execution, like methods in a Java class, and are written using a lower-case letter in the first position. Figure 2b shows the resulting AST when parsing the sample query.

3.3 AST Manipulation

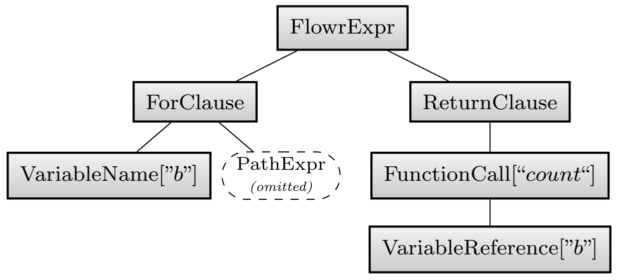
After the generated parser has processed the input query, the first AST instance is constructed using the rules given above. Several ASTs are generated and parsed in different

[1]http://www.antlr.org/

(a)

```
for $b in doc("dblp.xml")//article
return count($b)
```

(b)



(c)

```
expr:
  flowrExpr | functionCall | varRef ;
flowrExpr:
  "for" $ qname "in" expr
    { insertVariableReference(); }
  "return" expr
  -> ^(FlowrExpr
       ^(ForClause VariableName[qname] expr)
       ^(ReturnClause expr)
     );
functionCall:
  qname "(" expr ("," expr)* ")"
  -> ^(FunctionCall[qname] expr+);
qname:
  ("a".."z")+ ;
```

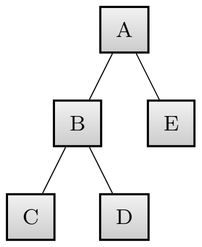
Figure 2: (a) A sample XQuery expression, (b) the AST generated for it, and (c) the grammar rules to transform the query into an AST

stages throughout the pipeline. ANTLR provides powerful extensions to the grammar rules, enabling AST-based query processing and optimization. For instance, the insertion of actions in arbitrary places of a rule (syntax: `{}`) and the specification of rewrite rules (syntax: `->`). In the following, additional extensions will be described, and Section 3.4 shows how to apply these techniques in a grammar-based query processor.

Parsing an AST is obviously an essential functionality for further manipulations. Therefore, the AST is serialized into a sequence of tokens. The serialization method implemented by ANTLR adds two special tokens to signalize descending and ascending movements—**DOWN** and **UP**—and scans the tree in pre-order. Figure 3a illustrates a sample tree and 3b its serialized token sequence. Note that the serialization is equivalent to the tree syntax in grammar rules, but using the **UP/DOWN** tokens instead of parenthesis: `^(A ^(B C D) E)`

Because *tree parsing* allows to manipulate an AST by reorganizing or replacing entire subtrees, we also use it for the *application of rewriting rules.* A crucial task in query

(a)



(b)

```
A DOWN B DOWN C D UP E UP
```

Figure 3: (a) A sample AST and (b) the serialized list of tokens

processing, however, is to apply rewrite rules under distinct conditions, which are independent of the AST structure. As an example, the compiler may rewrite a node access to an index scan when generating a QEP, given the condition that an index for that access exists. Such conditions can be incorporated into the grammar using *semantic predicates*, as will be shown later.

Sometimes, it is cumbersome to write full grammars in order to perform simple AST manipulation tasks, because only a particular subset of the rules is relevant. Since a parser always recognizes all valid expressions of a language, all necessary rules must be specified when generating it. To overcome this problem, ANTLR implements the concept of a *Tree Pattern Matcher*. These are special kinds of parsers that recognize any token sequence and trigger embedded actions when a certain pattern contained in the grammar rules is found. Making use of this feature, a grammar, having only the patterns that require rewrite, is sufficient to generate a tree scanner that performs the desired manipulation.

3.4 Applying ASTs in the Pipeline

Table 1 sketches four examples for the application of grammar rules on ASTs. Each rule and the related ASTs are reduced to the bare minimum required for showing the AST rewrite effects.

1. **Normalization** for a FLOWR expression is applied to replace multiple **for** clauses by nesting single **for**s. Here, the rule's **forClause** and **returnClause** were omitted for simplicity.
2. **Index Matching** for a **Path Step** generates alternative plans, when suitable indexes are available. The example addresses child nodes of the context item named 'author'. After the transformation, this logical data access is represented by a physical operator, either a document index scan (**ElementDocumentScan**) or an element index scan (**ElementIndexScan**)—physical operators available in our XDBMS. This rule is using a *semantic predicate*—**isIndexAvailable()**.
3. **Join Reordering** rules help to generate alternative plans having different join orders. Thus, cost-based

plan evaluation may aim for the cheapest join order when applying cost estimations to all the alternatives. In the example, the nested **for** clauses of expression 1 and 2 are swapped because their context is independent (the semantic predicate (condition) **isForClauseIndependent()** has to be checked first).

4. **Predicate Pushdown** moves the **whereClause** of a FLOWR expression into a nested FLOWR expression. However, again a check **isWhereIndependent()** is required to verify that the where predicate can be eagerly used as a filter. Normally a nested FLOWR expression is correlated to the outer one and, thus, a non-blocking **whereClause** (e.g., blocking clauses contain a grouping or ordering expression) is usually specified for the correlated input visible for the nested one.

4. OPPORTUNITIES AND CHALLENGES

Integrating the AST-only query manipulation approach into the XTC system became possible due to the extension of tree pattern matching in ANTLR. However, the existing query compilation process, sketched in Section 2, is not only complex but also well-studied, optimized, and supported by a huge amount of test cases. To replace the entire compilation process, a step-by-step approach seems to be feasible. As a consequence, the new AST-only version may start with a rudimentary set of rules that can be manually handled. In this way, the correctness of the rule's impact can be proved and a suboptimal QEP can be built and executed. In the next steps, existing optimization rules (i.e., Java code working on the QGM) can be transformed into declarative rules for the AST. Eventually, we hope to obtain at least the same set of powerful rewrite rules and integrate them into our new query processing approach. A major challenge is to express the flexibility of rules written in Java with the concepts and features of a language tool while retaining the same semantics.

The AST representation entails certain limitations to the semantic rewriting power, when only tree syntax is exploited within the rules. For instance, nested bindings and dependencies cannot always be decoupled, although decoupled processing may be semantically correct. For instance, for the join reorder example of Section 3.4 which depends on the evaluation of the **forClauseIndependent()** condition, a semantically reordering may also be possible in case of an existing dependency by simply exchanging axis relationships (e.g., from child axis to parent axis). However, XQuery semantics for empty sequences has to be observed[2]. Although all necessary conditions are represented within the AST, it seems to be extremely challenging to describe generic rules covering all the (hidden) corner cases.

Another aspect of rule-based optimization is to control the set of applicable and meaningful rules. Because not all rules do always improve a given query but only increase the optimization budget, the *selection and usage* of rules may become an important aspect. Moreover, we hope to exploit the easy way of expressing (new) rules to figure out which rules maybe applied independently and which not. However, not only the "fastest" rule chain is interesting but also the impact of enabling or disabling certain rules and, thereby, disabling dependent rules automatically. Furthermore, classical problems will reappear such as cycle detection and prevention, search space pruning, and various search strategies (e.g., bottom-up, top-down, etc.).

Eventually an AST-represented plan needs to be translated into a QEP that is often tailored to a specific (X)DBMS. Here again, rule-based translation allows to easily translate AST expressions into DMBS-dependent operators or code (see Table 1: example 2). In a perfect solution, it is sufficient to specify the set of available DBMS operators to automatically select the set of rules that produce a tailored QEP. Thus, a generic query optimization may be used for different DBMSs without losing expressiveness while reducing the efforts of optimizer development to a single instance.

Because XQuery is a Turing-complete language, its not possible to cover all language aspects within the AST grammar rules. For this reason, we want to explore the limitations and opportunities for a smooth integration.

Once the essential parts of the query processing pipeline are specified based on the AST concept, we can start with other aspects of optimization. For instance, the nice way of specifying query rewrite rules may help to easily identify opportunities for (intra) query parallelization.

5. SUMMARY & OUTLOOK

Based on the current state of XTC query processing—the classical approach sketched in Section 2—, we presented our first experience using only language tools. This lightweight description style of query translation and transformation seems to provide equal expressiveness as hand-written program code. However, up to now, neither all limitations are identified nor all opportunities disclosed.

We hope to reduce the complexity of writing query compilers and query optimizers while not affecting the quality of optimization. Furthermore, we focus on proving rule correctness, finding optimal rule sets, and identifying DBMS-independent approaches.

The next steps will deal with XQuery language aspects and physical operator mapping capabilities in XTC, before we want to compare the performance (i.e., overhead) of tailored program code with our new AST rule-based approach.

6. REFERENCES

[1] D. Chamberlin et al., *XQuery 1.1: An XML Query Language. W3C Working Draft*, (2008).

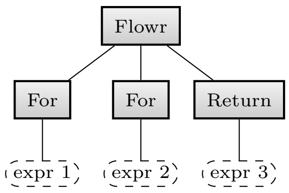
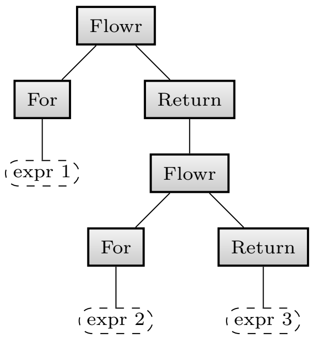
[2] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh, *Extensible query processing in starburst*, SIGMOD Rec. **18** (1989), no. 2, 377–388.

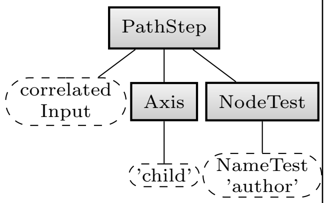
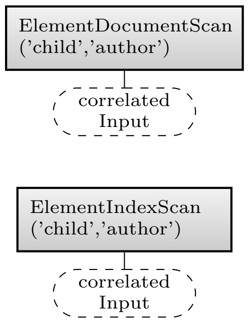
[3] Michael P. Haustein and Theo Härder, *An Efficient Infrastructure for Native Transactional XML Processing*, DATAK **61** (2007), no. 3, 500–523.

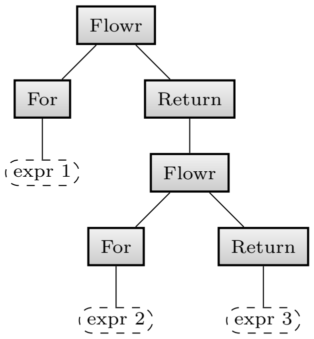
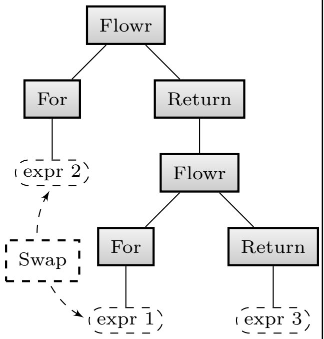
[4] Christian Mathis, *Storing, Indexing, and Querying XML Documents in Native Database Management Systems*, Ph.D. thesis, University of Kaiserslautern, München, 7 2009.

[2]When child nodes do not contribute to the evaluation, an empty sequence is required.

Rule	AST before	AST after

`duplicatedForClause:`
`  ^(Flowr`
`      forClause`
`      forClause`
`      returnClause`
`  )`
`  -> ^(Flowr`
`        ^(For`
`            forClause{1}`
`        )`
`        ^(Return`
`            ^(Flowr`
`                ^(For`
`                    forClause{2}`
`                )`
`                ^(Return`
`                    returnClause`
`                )`
`            )`
`        )`
`    )`
`;`
		 Flowr
		For
		Return
		expr 1
		Flowr
		For
		Return
		expr 2
		expr 3

`elementAccess:`
`  ^(Step`
`      contextItem`
`      Axis`
`      nodeTest`
`  )`
`  -> { isIndexAvailable }?`
`      ^(ElementIndexScan[`
`            Axis,`
`            nodeTest`
`        ]`
`        contextItem`
`    )`
`  -> ^(ElementDocumentScan[`
`            Axis,`
`            nodeTest`
`        ]`
`        contextItem`
`    )`
`;`
	 PathStep
	correlated Input
	Axis
	NodeTest
	'child'
	NameTest 'author'
		 ElementDocumentScan ('child','author')
		correlated Input
		ElementIndexScan ('child','author')
		correlated Input

`joinReorder:`
`  ^(Flowr`
`      forClause           // 1`
`      ^(ReturnClause`
`          ^(Flowr`
`              forClause   // 2`
`          )`
`      )`
`  )`
`  -> { isForClauseIndependent }?`
`   ^(Flowr`
`      forClause{2}`
`      ^(ReturnClause`
`          ^(Flowr`
`              forClause{1}`
`          )`
`      )`
`   );`
	 Flowr
	For
	Return
	expr 1
	Flowr
	For
	Return
	expr 2
	expr 3
		 Flowr
		For
		Return
		expr 2
		Flowr
		Swap
		For
		Return
		expr 1
		expr 3

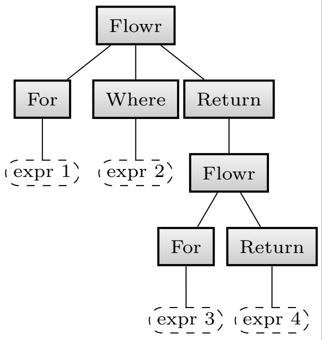
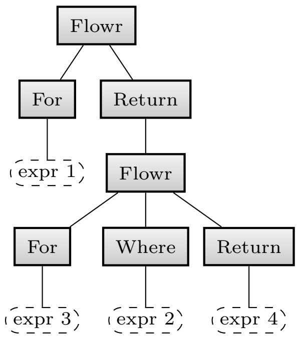
`predicatePushDown:`
`  ^(Flowr`
`      forClause`
`      whereClause`
`      ^(ReturnClause`
`          ^(Flowr`
`              forClause`
`              returnClause`
`          )`
`      )`
`  )`
`  -> { isWhereIndependent }?`
`  ^(Flowr`
`      forClause`
`      ^(ReturnClause`
`          ^(Flowr`
`              forClause`
`              whereClause`
`              returnClause`
`          )`
`      )`
`  );`
	 Flowr
	For
	Where
	Return
	expr 1
	expr 2
	Flowr
	For
	Return
	expr 3
	expr 4
		 Flowr
		For
		Return
		expr 1
		Flowr
		For
		Where
		Return
		expr 3
		expr 2
		expr 4

Table 1: Examples of AST manipulation for XQuery processing