Querying versioned software repositories

Christopeit, Dietrich ; Böhlen, Michael H ; Kanne, Carl-Christian ; Mazeika, Arturas

Abstract: Large parts of today's data is stored in text documents that undergo a series of changes during their lifetime. For instance during the development of a software product the source code changes frequently. Currently, managing such data relies on version control systems (VCSs). Extracting information from large documents and their different versions is a manual and tedious process. We present Qvestor, a system that allows to declaratively query documents. It leverages information about the structure of a document that is available as a context-free grammar and allows to declaratively query document versions through a grammar annotated with relational algebra expressions. We define and illustrate the annotation of grammars with relational algebra expressions and show how to translate the annotations to easy to use SQL views.

DOI: https://doi.org/10.1007/978-3-642-23737-9_4

Querying Versioned Software Repositories

Dietrich Christopeit[1], Michael Böhlen[2],
Carl-Christian Kanne[3], Arturas Mazeika[4]

[1]christo@ifi.uzh.ch, [2]boehlen@ifi.uzh.ch,
[3]kanne@informatik.uni-mannheim.de, [4]amazeika@mpi-inf.mpg.de

Abstract. Large parts of today's data is stored in text documents that undergo a series of changes during their lifetime. For instance during the development of a software product the source code changes frequently. Currently, managing such data relies on version control systems (VCSs). Extracting information from large documents and their different versions is a manual and tedious process. We present Qvestor, a system that allows to declaratively query documents. It leverages information about the structure of a document that is available as a context-free grammar and allows to declaratively query document versions through a grammar annotated with relational algebra expressions. We define and illustrate the annotation of grammars with relational algebra expressions and show how to translate the annotations to easy to use SQL views.

1 Introduction

Modern software engineering tools process large repositories of source code to assist software developers and analysts with the code retrieval as well as with the computation of various metrics over the source code. Frequently, such tools use handcrafted custom code to extract information and compute metrics. This is tedious, error-prone and brittle. Some approaches offer efficient but very specialized and limited querying capabilities (retrieve a given version of a file), other approaches offer general but hard to formulate and inefficient querying capabilities (extract lines of code from all files satisfying a regular expressions), or yet other easy to formulate, efficient, but only predefined querying capabilities (extract all names of functions from the versions of the source).

In this paper we propose Qvestor (querying versioned software repositories), a prototype implementation of a software query and analysis tool that (i) offers a general querying interface, (ii) allows a natural and easy way to formulate queries, and (iii) answers queries efficiently. Qvestor (i) parses the source of the documents using context-free grammars, (ii) allows to formulate queries declaratively using the components of the grammar (i.e., both the semantics and specifics of the code), and (iii) uses database query optimization techniques. This yields a general (applies for any data that adheres to a predefined grammar), elegant, concise (expresses in relational algebra operators), yet efficient (allows easy optimizations) approach to extract structured data from repositories. This enables a higher degree of reuse, and leverages database query processing techniques to the analysis of source code.

The development of QVESTOR is subtle and requires to integrate technologies and concepts from both compiler theory (to extract and store the data in relations) and database theory (to define a query language, capabilities, and show how to answer queries over extracted data). We use annotations from compiler theory as a means to formulate and execute queries. Similar to compiler theory, our annotations are program codes that are assigned to the alternatives of the rules of the grammar that are executed once the alternative is selected. In contrast to compiler theory, we associate relations with the alternatives and express annotations declaratively. Query execution then iteratively executes these queries starting with the bottom annotations (i.e., annotations over terminal relations) and finishing with the top annotations in the grammar.

The paper is organized as follows: In Section 2 we discuss the related approaches. In Section 3 we introduce our running example and discuss compiler and database essentials in the context of our approach. In Section 4 we set the foundation and present the building blocks for our system. Then declarative querying (Section 4) with the help of annotations of the leaves and propagation and combination of results in inner nodes is given. In Section 5 we outline the architecture of QVESTOR and sketch the algorithm to transform a list of grammar annotations into SQL view definitions. Finally, we conclude and offer future work in Section 6.

2 Related Work

Krishnamurthy et al. developed SystemT [7]. This system uses a declarative query language AQL to extract information from blog-entries in natural language. With AQL it is possible to express grammar rules in an SQL-like style that describe what the user wants to retrieve (from the blogs). The authors found out that parsing a document is costly and therefore applied rewrite methods to reduce execution costs. In our system we also try to avoid the parsing of data if not needed. In contrast to SystemT we want to focus in QVESTOR on querying versioned data.

Fischer et al. [4] retrieve information from version control data and Bugzilla Bug reports to analyze software evolution. By making this information available in an SQL database simple code evolution queries are possible. However, these queries merely use regular expressions rather than being able to query the code itself. In [6] Kemerer et al. use information from change logs to compute statistical information about software changes. The approach does not query over changes but merely computes the basic statistics. In our system such queries can be formulated declaratively much easier. In addition, our system is not limited to the basic statistics but also allows any sophisticated queries expressible through annotations and the grammar of the source code. Solutions for flexible querying of the source code are proposed by Paul et al. [9]. Our system differs from their SCA algebra and ESCAPE system in the way that for each queryable source code component there must be an object definition for the (OO) data store. Thus the possible queries are to some extend limited by the specification

of the objects of the data store. Furthermore, our system aims to reuse as much existing query facilities as possible. Therefore, we chose to use relational algebra expressions that are attached to the grammar specification of the data to be queried. While in ESCAPE new object definitions would have to be specified if the granularity of the queries changes we would just have to change grammar annotations.

Chen et al. [3] describe a system for C code analysis using relational databases. Unlike our proposed system the CIA system does not facilitate a tight coupling between the database, declarative querying, and versions of the source code. For CIA several tools must preprocess the files and store the processed information in a database for declarative querying. Query capabilities are limited by the preprocessing part. In our system tight coupling enables us to query syntax and semantic of source code files directly in the database and does not limit results. Other systems like Rigi [8] or SHriMP [10] mainly focus on visualization of program source dependencies.

Abiteboul et al. [1] describe how semi-structured documents can be queried in an OO-DBMS using a grammar for the document's structure. While queries that involve structural elements are mostly rewritten and parts of the query are pushed into the grammar automatically as annotations we aim for an approach where the user can manage the annotations of the parse tree. In [2] querying of XML documents is described on the native XML-DBMS Natix[5]. While we want to support a wide range of semi-structured versioned data describable by grammars we also want to translate our grammar annotations to generate views that can easily be used in queries.

3 Running Example, Grammar, Parse Trees, and DB Schema

3.1 Example

Our running example consists of three versions of the C-like source code. We start (version one) with a very basic function, which evolves (version two) into another function, which, in turn, evolves into an even more complicated version (version three) of nested loops.

The example (see Figure 1) represents key aspects of evolving code. It consists of evolving signature of functions (version one), change in function calls (version two), additions and deletions of new functions (version three). In this paper we show how to declaratively query the evolving code for all such constructs.

3.2 Grammar

The grammar establishes a structure for the (otherwise unstructured) versioned documents. In addition to general substring queries over unstructured versioned documents, this allows to formulate queries related to concepts of the code. For C code, for example, one can formulate queries involving variables, functions,

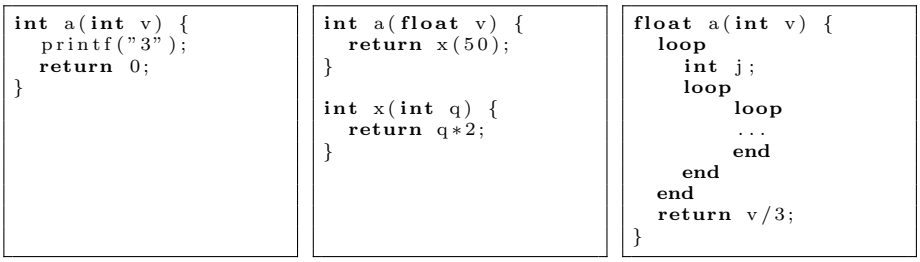


Fig. 1. Three versions of C-style like source code.

and specific statements of the language. Every rule ($r_i ::= u_{i,0} \mid u_{i,1} \mid \ldots \mid u_{i,m}$) consists of the left hand side (abbreviated LHS; e.g. r_i), the right hand side (abbreviated RHS; e.g. $u_{i,0} \mid u_{i,1} \mid \ldots \mid u_{i,m}$) and the assignment symbol (::=) that divides the rule into the LHS and RHS. The LHS introduces a new identifier r_i; The RHS defines the rule for the new identifier. In the most general form, the RHS consists of alternatives ($u_{i,j}$) separated by delimiter |. Every alternative $u_{i,j}$, in turn, consists of components $u_{i,j} = \$1_{i,j} \ldots \$n_{i,j}$. We use the simplified notation for the components $u = u_{i,j} = \$1 \ldots \$n$ whenever it is clear from the context which components we are referring to.

The components of the alternatives may be either terminal symbols, identifiers defined by other rules or the current rule (in this case the rule is called recursive), or be the empty symbol ε.

To use a specific attribute of grammar component k we use the dot (.) to access the attribute of the component. For example, $\$k$.`C` refers to the content of the component and $\$k$.`P` refers to the parent of the component.

Consider, for example, rule

$$\begin{aligned} \text{expr} ::= &\ \text{expr '*' expr} \mid \text{expr '/' expr} \mid \text{IDENT} \mid \text{fnCall} \mid \text{const} \\ = &\ u_{i,0} \mid u_{i,1} \mid u_{i,2} \mid u_{i,3} \mid u_{i,4}. \end{aligned}$$

The rule has five alternatives; the components of the 0^{th} alternative are $u_{i,0} =$ \$1 \$2 \$3, where $\$1 = \text{expr}, \$3 = \text{expr}$ and $\$2 =$ '*'.

Table 1 summarizes the grammar used in the running examples of the simplified C code .

3.3 Parse Trees

The parse tree of the software code represents the syntactic structure and elements of the source code. The parser builds the parse tree by recursively applying the grammar over the source code. The parser recursively tries to match rules by starting to match sequences of terminals. If such a sequence matches a rule it creates a node representing the LHS of the rule and attaches as leaves the

Rule No	Rule
0	start :: = fnDefLst
1	fnDefLst :: = fnDef fnDefLst | fnDef
2	fnDef :: = type IDENT '(' varDecl ')' '{' stmtLst 'return' expr ';' '}'
3	type :: = INT | FLOAT
4	varDecl :: = type IDENT';' | type IDENT '=' const';'
5	stmtLst :: = stmt ';' stmtLst | stmt ';' | ε
6	stmt :: = fnCall | varDecl | loopStmt
7	fnCall :: = IDENT '(' const ')'
8	const :: = INTCONST | STRCONST
9	expr :: = expr '*' expr | expr '/' expr | IDENT | fnCall | const
10	loopStmt :: = 'loop' stmtLst 'end'

Table 1. The grammar of the simplified C code used throughout the paper

nodes representing the terminals. The same is done if a rule matches a sequence of RHS nodes that are used in another rule. Step by step the parser builds a so-called parse tree bottom-up.

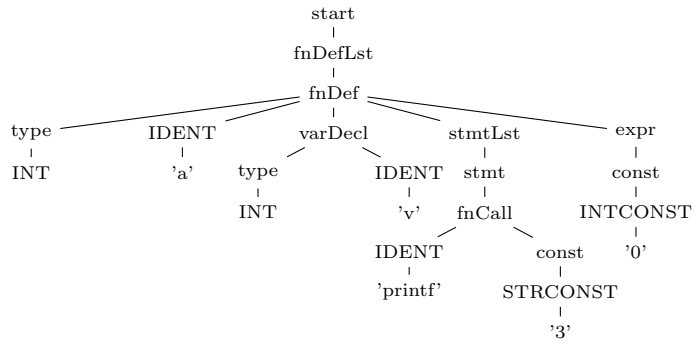
Consider, for example, the source code of Version 1 in Figure 1 and grammar in Table 1. The parser starts with matching '`int`' to the terminal `INT` in rule 3, creates a '`type`' node and proceeds with '`a`' as '`IDENT`'. Now the parser has two choices. It can create a node '`varDecl`' if the next symbol is '`;`' and attach '`type`' and '`IDENT`' as children. On the other hand, if the next symbol is '`(`' it can proceed with matching rule 2 – eventually creating a node '`fnDef`'. This is the case here. The process is continued in bottom-up manner until all the source code is processed. The resulting parse tree is shown in Figure 2(a) (the complete parse trees for all versions are shown in Figure 2).

3.4 Terminal Relations

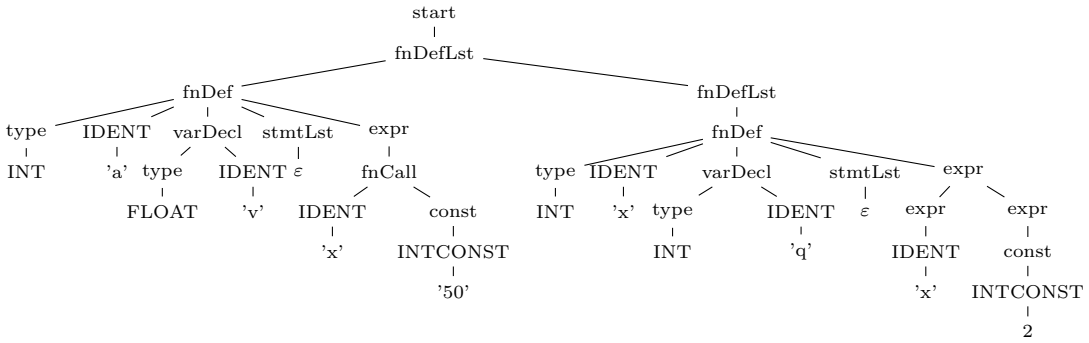
With every grammar component (like IDENT, INT, FLOAT or expr) in the grammar we associate a relation and store the tuples related to the component in the associated relation. The schema of the relations consists of the following attributes : `ID` (uniquely identifies the node), `V` (code version), `L` (line number in the document; each word is in a separate line), `C` (content of terminal symbols), `N` (name of rule), `S` (next right sibling, and next right component in grammar rule) and `P` (parent ID of the node). This allows to describe and fully reconstruct the version trees from the table. We reference the relations in the following way. Let $\$k_{i,j}$ refer to the k^{th} component of the j^{th} alternative of the i^{th} rule. If the component is a terminal symbol then the associated relation is referred by $\tau_{k,i,j}$.

For example consider relation $\tau_{1,7,0}$ (i.e. relation for $\$1_{7,0}$ component). For our running example the result is given in Table 2.

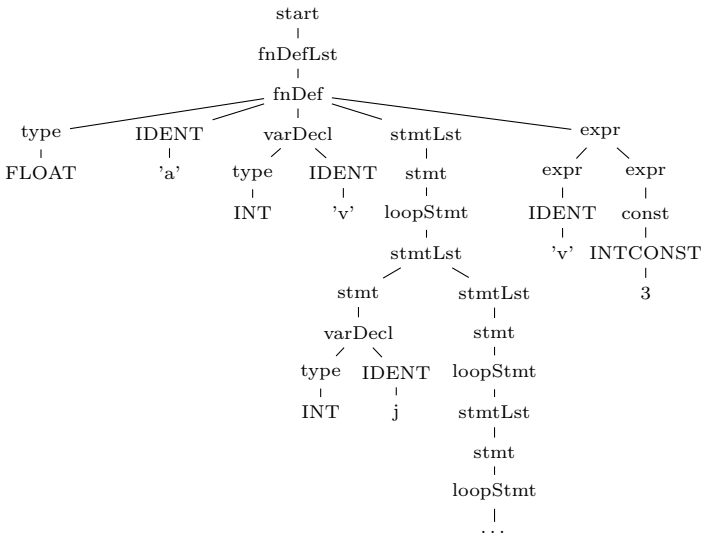
If the component is a non-terminal then the associated relation is referred by $t_{i,j}$.



(a) Version 1



(b) Version 2



(c) Version 3

Fig. 2. Parse trees

ID	V	L	C	N	S	P
7	1	8	printf	IDENT	-	10
8	2	9	x	IDENT	-	11

Table 2. Relation of the components of the alternatives of the grammar rules

4 Declarative Querying

4.1 Data at Leaves and Data at inner Nodes

In this paper we query data that is associated with nodes in the parse tree. Conceptually we distinguish between two types:

Definition 1 (Node Data). *While the parser builds up the parse tree certain information is collected for every node – leaf node or inner node:*

- `ID` *(uniquely identifies the node)*
- `V` *(code version)*
- `L` *(line number in document; one word per line)*
- `C` *(content of terminal symbols)*
- `N` *(name of rule)*
- `S` *(next right sibling, next right component in grammar rule)*
- `P` *(parent ID of the node)*

The set of all ***node data*** *of a given parse tree is denoted by* $\mathcal{ND}$.

To formulate declarative queries in our system we use the node data of different nodes, combine it with relational algebra operators to get **composed data**.

Definition 2 (Composed Data). *Let* k *be a node in the parse tree that has* $n \in \mathbb{N}_0$ *children* $c_0, \ldots, c_n$. *Let node* k *further be a node that is the RHS of rule* k *option* j. *Let* $d_k \in \mathcal{ND}$ *be the* ***node data*** *of node* k *and* $d_{c_0}, \ldots, d_{c_n}, d_{c_i} \in \mathcal{ND}, 0 \leq i \leq n$ *be the node data of the children of* n. *Let* $expr_i, 0 \leq i \leq n$ *and* $expr_k$ *be relational algebra expressions.*

Without loss of generality let $c_0, \ldots, c_n$ *be leaves. The* ***Composed Data*** *is calculated for one version like the following for:*

- c_i*:* $expr_i(d_{c_i}) = \tau_{i,k,j}$
- k*:* $expr_k(expr_0(d_{c_0}), expr_1(d_{c_1}), \ldots, expr_n(d_{c_n}), d_k) = t_{k,j}$

If there exist $m, m \in \mathbb{N}$ *versions then:*

- $\tau_{i,k,j} = \bigcup_m expr_i(d_{c_i})$
- $t_{k,j} = \bigcup_m expr_k(expr_0(d_{c_0}), expr_1(d_{c_1}), \ldots, expr_n(d_{c_n}), d_k)$

$\tau_{i,k,j}$ *and* $t_{k,j}$ *are the union of the* $\tau_{i,k,j}$ *and* $t_{k,j}$ *of the individual versions.*

In Figure 3 the situation is shown for node k with two children c_0 and c_1.

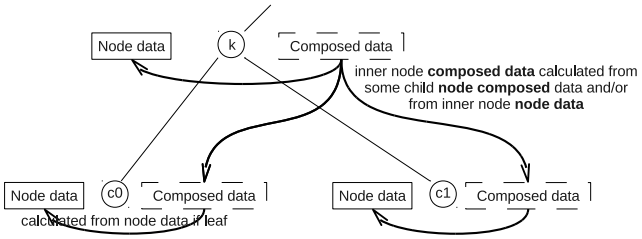


Fig. 3. Data attached to Nodes in Parse Tree.

4.2 Annotations

In compiler theory, annotations are program codes that are assigned to the alternatives ($u_{i,j}$) of the rules. Once the alternative is selected during build up of the parse tree, the corresponding annotation is executed, a result is calculated and/or output. Therefore, annotations can be viewed as ways to both formulate and answer queries over the source code. In this paper we focus on the declarative capabilities of the query formulations of annotations. Since we use database operations including selection, projection, and join, existing database techniques are applied to optimize and answer such complicated queries.

All our annotations are expressed in terms of the node data (see Definition 1) and composed data (see Definition 2). For example to access all `IDENT` nodes in expressions (`expr`) and print their content (`C` attribute) we need to formulate and execute this query:

$$\mathit{\Pi}_{\tau_{1,9,2}.C}(\tau_{1,9,2})$$

This example accesses the properties of only one node and does not require any joins. Consider an example now, when we want to select all variable names which get assigned a constant 5. These are the names of all `IDENT` nodes (cf. alternative 1, rule 4 in Table 1) and all `CONST` nodes (cf. alternative 1, rule 4 in Table 1) such `IDENT` and `CONST` have the same LHS `varDecl`.

$$\mathit{\Pi}_{\tau_{2,4,1}.C}(\tau_{2,4,1} \bowtie_{\tau_{2,4,1}.P=\tau_{4,4,1}.P} (\sigma_{\tau_{4,4,1}.C=5}(\tau_{4,4,1})))$$

In general, our annotations allow the following operators and predicates:

- Selection σ_P, projection π_P, join $\bowtie_P$, cartesian product $\times$, renaming ρ_V, set operators $(\cup, \cap, -)$, and aggregation ϑ.
- Schema identifier $\mathbf{S}(A)$: denotes the schema of a relation A
- Schema modification: addition of an attribute $\circ$ e.g. $\mathbf{S}(A) \circ C$; and removal of the attribute $-$ e.g. $\mathbf{S}(A) - C$
- Predicates: $<, >, =, \leq, \geq, \neq$

We describe the declarative querying and formulation annotations in turn. First we show how to formulate annotations over the leaf nodes, then we generalize it for all nodes, and eventually we explain how to use relational algebra operators in the annotations.

4.3 Annotating the Leaves

An annotation over a leaf node applies the given relational algebra operators over the associated relation and returns the relation. The schema of the returned relation solely depends on the operators applied over the source relation. Since the source relation is always (`ID, V, L, C, N, S, P`) (see Section 3.4 and Definition 1) the selection over the relation returns the relation of the same schema. Similarly set union, difference, and intersection based on the relation does not change the schema. In contrast, other relational algebra operations change the schema (return larger, smaller, or renamed schemata). For example, the projection operator (usually) reduces the number of attributes in the schema, while (self-) join and Cartesian product doubles the number of attributes in the schema. The general form of the annotation over the leaf node is of the form:

$$t_{i,j} = op_1(\dots op_n(\tau_{k,i,j})\dots), \tag{1}$$

where $op_1, \dots, op_n$ are relational algebra operators, and $\tau_{k,i,j}$ is the terminal relation of the alternative of the rule; relational operators may use only the attributes of the current terminal relation. The resulting relation is called $t_{i,j}$ and associated with the alternative $u(i, j)$.

Consider, for example, the following annotation:

$$t_{7,0} = \Pi_V(\sigma_{C='printf'}(\tau_{1,7,0}))).$$

Then the annotation selects all versions of the source code that calls function `printf`. The following table is returned as the answer to this query:

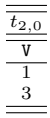
$t_{7,0}$
V
1

4.4 Annotating the non-Leaves

Very similar reasoning applies for the annotations of the non-leaves including the relational algebra operators and the schema of the returned result. The key difference is that now the relational algebra operators include the relations and information from one level (in terms of parse tree/components of the grammar rules) below the relation it is formulated at. For example, consider the query that selects all the versions of the functions that have return parameter of type integer (`INT`). This results in the following query:

- $t_{2,0} = \Pi_{\tau_{2,2,0}.C}(t_{3,0} \bowtie_{t_{3,0}.P=\tau_{2,2,0}.P} \tau_{2,2,0})$
- $t_{3,0} = \tau_{1,3,0}$

The answer of the query is the following relation:



4.5 General Queries of Software Repositories

A general query over software repository consists of a set of annotations such that (i) every annotation is a proper annotation either of leaves or non-leaves (see Sections 4.3 and 4.4) and (ii) if a node is annotated then there must exist a path to a leaf such that every node on a path is annotated.

4.6 Use Case: Find all Versions of the Software that have Loops of Depth three or higher

Identifying the software versions that have feature nested loops of depth of at least 3 is interesting both conceptually and technically. Conceptually, this can indicate versions that have performance issues: nested loops of depth of at least 3 mean at least cubic complexity and may call for attention. Technically, this is a challenge, because in principal, annotations employ context free grammars and expressing such constructs show the power of the language.

Rule No	Rule	Annotation
0	start :: = fnDefLst	
1	fnDefLst :: = fnDef fnDefLst | fnDef	
2	fnDef :: = type IDENT '(' varDecl ')' '{' stmtLst 'return' expr ';' '}'	$\pi_{t_{5,0}.V}(\sigma_{t_{5,0}.D \geqslant 3}(t_{5,0}))$
3	type :: = INT | FLOAT	
4	varDecl :: = type IDENT';'	$\rho_{\mathbf{S}(\tau_{2,4,0})\circ D}(\pi_{\mathbf{S}(\tau_{2,4,0})\circ 0}(\tau_{2,4,0}))$
	| type IDENT '=' CONST';'	$\rho_{\mathbf{S}(\tau_{2,4,1})\circ D}(\pi_{\mathbf{S}(\tau_{2,4,1})\circ 0}(\tau_{2,4,1}))$
5	stmtLst :: = stmt ';' stmtLst	$\big(t_{6,0} \cup t_{6,1} \cup t_{6,2}\big) \cup t_{5,0}$
	| stmt';'	$t_{6,0} \cup t_{6,1} \cup t_{6,2}$
	$\mid \varepsilon$	
6	stmt :: = fnCall	$t_{7,0}$
	| varDecl	$t_{4,0} \cup t_{4,1}$
	| loopStmt	$t_{10,0}$
7	fnCall :: = IDENT '(' const ')'	$\rho_{\mathbf{S}(\tau_{1,7,0})\circ D}\pi_{\mathbf{S}(\tau_{1,7,0})\circ 0}(\tau_{1,7,0}))$
8	const :: = INTCONST | STRCONST	
9	expr :: = expr '*' expr | expr '/' expr | IDENT | fnCall| const	
10	loopStmt :: = 'loop' stmtLst 'end'	$\rho_{\mathbf{S}(t_{5,0})\circ D}$ $(\pi_{\mathbf{S}(t_{5,0})-D\circ \Im_{MAX(t_{5,0}.D)+1}}(t_{5,0}))$

Table 3. Annotation query (along with the grammar rules)

Conceptually, we formulate this query in the following way. We access all the statements in the code. The non-loop statements get the depth attribute assigned to zero. The loop statements compute their depth in the following way.

Let s_l be a loop statement, and S_e be the set of statements s_l encloses. Let m be the maximum depth of s_l statements. Then s_l gets $m + 1$ depth.

The annotations of the query are shown in Table 3. Rules 4 and 7 define the depth of non-loop statements. Rule 10 defines the depth of the loop statements. Rules 2, 5, 6, and 7 combine the result. The following is the answer to the query for our running example:

$$\frac{\frac{\frac{t_{2,0}}{\mathtt{V}}}{3}}{}$$

5 Implementation of QVESTOR

5.1 Architecture

We propose and implemented a modular system to answer such declarative queries. Our system consists of the following key modules: declarative query, query rewriter, SQL database, and versions of some programming language source code. The *declarative query* module inputs the query from the user. This is expressed in terms of the rules and attributes of the grammar. Then the *query rewriter* inputs the declarative query and transforms it into an SQL query over the database of versions with the help of *annotated grammar*. The detailed architecture of the system is depicted in Figure 4.

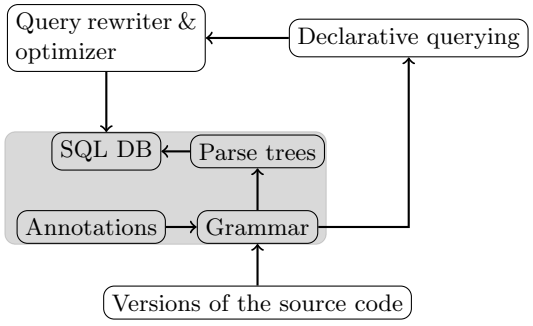


Fig. 4. Architecture of the system

We were reusing expertise and components from the state-of-the-art and standard systems as much as possible. For example, the parse trees were stored in the PostgreSQL database; user defined grammar annotations were translated into database views. This allowed to both achieve efficient storage of the data and obtain efficient query execution plans from query optimizers.

5.2 DB Schema

We store all our data in an SQL database. This allows us to reuse most of the database functionality including ease of expression of queries, query optimization, and effective and efficient storage of the data.

On the database level we keep all the data in the `PARSETABLE` table. The schema of the table is the same as the schema of the relations of the alternatives of the rules (see Section 3.4), while the table integrates all the node data (see Definition 1).

The tuples for Version 1 (Figure 2(a)) of our running example are given in Table 4.

ID	V	L	C	N	S	P
19	1			start		
18	1			fnDefLst		19
17	1			fnDef		18
2	1			type	3	17
1	1	1	'int'	INT		2
3	1	2	'a'	IDENT	7	17
7	1			varDecl	13	17
5	1			type	6	7
4	1	4	'int'	INT		5
6	1	5	'v'	IDENT		7
13	1			stmtLst	16	17
12	1			stmt		13
11	1			fnCall		12
8	1	8	'printf'	IDENT	10	11
10	1			const		11
9	1	10	'3'	STRCONST		10
16	1			expr		17
15	1			const		16
14	1	15	'0'	INTCONST		15

Table 4. `PARSETABLE`

5.3 Query Formulation, Translation, and Execution

The implemented system allows for the user to formulate the queries in the SQL-like language. This allows all declarative constructs of the SQL including the `SELECT`, `FROM`, `WHERE`, `GROUP BY` clauses and all supported predicates of PostgreSQL. In addition we allow the use of symbols $\tau_{k,i,j}$ and $t_{i,j}$ as relation names in the queries (see Section 4). Due to the space constraints we do not define the extended language and hope that the reader gets the spirit of the language.

Given a formulated query in our SQL-like language, we transform it into SQL and send it to PostgreSQL for optimization and execution. The translation of the formulated query is basically achieved in two steps. First, we scan all queries and replace annotations using $\tau_{k,i,j}$ and $t_{i,j}$ with select queries over the `PARSETABLE`(s) with `WHERE` clauses. Second, we scan all queries for the second time and create views in the query statements.

As an example consider the query that retrieves the names of the functions that return an integer type. The following is the query expressed with the help of annotations:

- $t_{3,0} = \tau_{1,3,0}$
- $t_{2,0} = \Pi_{\tau_{2,2,0}.C}(t_{3,0} \bowtie_{t_{3,0}.P=\tau_{2,2,0}.P} \tau_{2,2,0})$

This query should be formulated in the following SQL-like way:

- $t_{3,0}$ = `SELECT * FROM` $\tau_{1,3,0}$;
- $t_{2,0}$ = `SELECT` $\tau_{2,2,0}$`.C FROM` $t_{3,0}$, $\tau_{2,2,0}$ `WHERE` $t_{3,0}$`.P =` $\tau_{2,2,0}$`.P`;

After the 1st stage the queries become:

```
SELECT p.ID AS ID,
  k.V AS V, k.L AS L, k.C AS C,
  K.N AS N, K.S AS S, p.P AS P
FROM PARSETABLE k, PARSETABLE p
WHERE k.P = p.ID AND
  k.N = 'INT' AND
  p.N = 'type';
```

```
SELECT k.C AS C
FROM t3,0 t, PARSETABLE k,
  PARSETABLE p
WHERE t3,0.P = k.P AND
  t3,0.P = p.ID AND
  k.P = p.ID AND
  p.N = 'fnDef' AND
  k.N = 'IDENT';
```

After the 2nd stage the queries become:

```
WITH t3,0 AS (
SELECT p.ID AS ID,
  k.V AS V, k.L AS L, k.C AS C,
  K.N AS N, K.S AS S, p.P AS P
FROM PARSETABLE k, PARSETABLE p
WHERE k.P = p.ID AND
  k.N = 'INT' AND
  p.N = 'type'),
```

```
t2,0 AS (
SELECT k.C AS C
FROM t3,0 t, PARSETABLE k,
  PARSETABLE p
WHERE t3,0.P = k.P AND
  t3,0.P = p.ID AND
  k.P = p.ID AND
  p.N = 'fnDef' AND
  k.N = 'IDENT');
```

6 Conclusions and Future Work

In this paper we presented a system to formulate and answer declarative queries over the versioned source code. Annotations of the grammar rules are the key that allows the declarative querying and connection of the components in the system: first, the annotations allow to naturally formulate queries over the source code (compared, for example, to regexp), and second, allow to translate the queries into SQL and answer them efficiently. Our system consists of declarative query, query rewriter, SQL database, and versions of source code. Tight coupling with the components from state-of-the-art database systems allowed for effectively and efficiently both store and query the data.

Future work will concentrate on the introduction of a sequence model for source code versions. This model will account for the fact that versions are not necessarily available in fully materialized but compressed form. We will introduce a model that describes code evolution with the help of differences between the pairs of versions. It is not necessary to generate a parse table for every version of the code depending on the user's query formulated through the annotations. By defining rules to rewrite the user's query and annotations we see potential for query optimization (e.g. save parsing of code versions that can not participate in the result).

As we have already implemented a (sub-) operator for the generation of the `PARSETABLE` we will further proceed with integrating the automatic view generation from grammar annotations by implementing an operator tightly into the DBMS. Currently this rather done with an external program than being an extension to SQL.

References

1. S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *VLDB '93*, pages 73–84, 1993.
2. V. Aguilera, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez. Querying xml documents in xyleme, 2000.
3. Y.-F. Chen, M. Nishimoto, and C. Ramamoorthy. The c information abstraction system. *IEEE Transactions on Software Engineering*, 16:325 – 334, 1990.
4. M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03*, 2003.
5. C.-C. Kanne and G. Moerkotte. Efficient storage of xml data. *ICDE'00*, December 1999.
6. C. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25:493 – 509, 7 1999.
7. R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. Systemt: a system for declarative information extraction. *ACM SIGMOMOD Record*, 37:33–44, 2008.
8. H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse-engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice*, 5:181 – 204, 1993.
9. S. Paul and A. Prakash. Querying source code using an algebraic query language. *Proceedings International Conference on Software Maintenance ICSM-94*, pages 127–136, 1994.
10. M.-A. Storey and H. Müller. Manipulating and documenting software structures using shrimp views. *Proceedings in Software Maintenance*, pages 275 – 284, 1995.