Part III

The WIQA Framework

The *WIQA - Information Quality Assessment Framework* is a set of software components for filtering information using different quality-based information filtering policies. The WIQA framework can be employed by applications which process information of uncertain quality and want to enable users to filter information using different policies. The framework has been designed to fulfill the following requirements:

Flexible Representation of Quality-Related Meta-Information. As we have seen in Chapter 3, information quality assessment relies on a wide range of different quality indicators. Which quality indicators are relevant depends on the application domain and the quality dimensions to be assessed. Important quality indicators in the context of web-based information systems are provenance information, ratings, and background information about information providers. The WIQA framework uses Named Graphs as a flexible data model for representing information together with quality related meta-information.

Support for Information Filtering Policies. The relevance of different quality dimensions and the metrics used to assess these dimensions depend on the application domain, the quality indicators available, the task at hand, and the subjective preferences of the information consumer. Therefore, information consumers use a wide range of different policies for determining whether to accept or reject information. The WIQA framework allows various policies to be employed for filtering information. Policies are expressed using a declarative policy language and can combine context-, content-, and rating-based quality assessment metrics.

Explaining Filtering Decisions. The accuracy of assessment results is often uncertain due to the limited availability of quality indicators and the often uncertain quality of the quality indicators themselves. Therefore, the final subjective decision of an information consumer, whether to trust or distrust assessment results, depends on his understanding of the quality indicators and the assessment metrics that have been used in the assessment process. In order to support information consumers in their trust decision, the WIQA framework can generate detailed explanations about filtering decisions.

Figure 8.1 gives an overview about the components of the WIQA framework and illustrates how applications interact with the framework. The WIQA framework consists of the *NG4J - Named Graphs API for Jena* and the *WIQA - Filtering and Explanation Engine.*

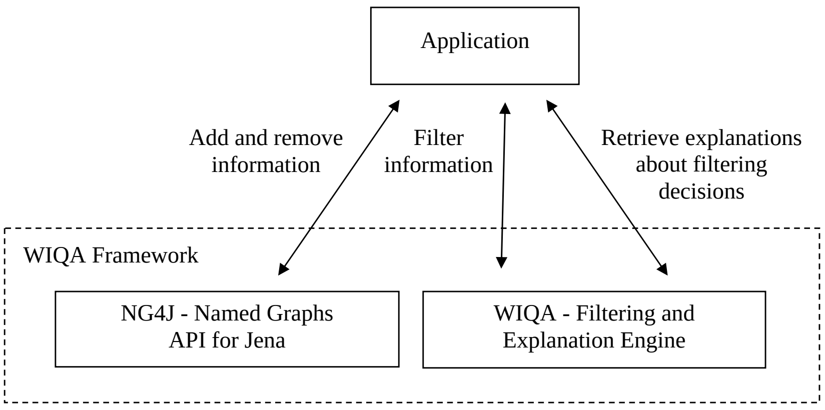
Application
Add and remove information
Filter information
Retrieve explanations about filtering decisions
WIQA Framework
NG4J - Named Graphs API for Jena
WIQA - Filtering and Explanation Engine

Figure 8.1: Overview of the WIQA framework.

NG4J - Named Graphs API for Jena is a software toolkit for creating, manipulating, persisting, and exchanging sets of named graphs. Graph sets can be stored in memory or in a relational database. The API provides parsers and serializers for the TriX, TriG, and RDF/XML syntaxes and allows graph sets to be signed using the Semantic Web Publishing Vocabulary. The WIQA framework uses NG4J to store information together with quality-related meta-information.

The *WIQA - Filtering and Explanation Engine* determines the subset of the triples contained in a set of named graphs that match a given filtering policy. Filtering policies are expressed using the WIQA-PL policy language. Applications present a set of named graphs and a WIQA-PL policy to the WIQA - Filtering and Explanation Engine. Based on the policy, the engine generates a view on the graph set containing all triples that fulfill the policy. This *set of accepted triples* is returned to the application. Figure 8.2 illustrates the process of promoting triples from the set of named graphs into the set of accepted triples.

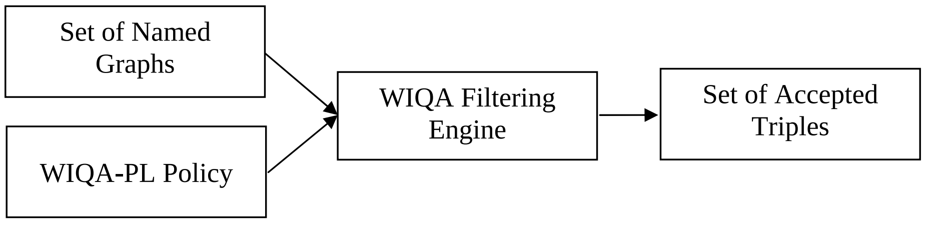

Figure 8.2: Overview of the filtering process.

The WIQA - Filtering and Explanation Engine can generate explanations about filtering decisions. An application can present an accepted triple to the engine which returns an explanation why the triple satisfies the given policy. The engine can generate two types of explanations: Textual explanations and RDF explanations. Textual explanations can be displayed directly to the end-user. RDF explanations may be used by the application for further processing.

The following chapters describe the WIQA framework in detail:

Chapter 9: Expressing Information Filtering Policies. Within the WIQA framework, information filtering policies are expressed using the WIQA-PL policy language. This chapter introduces the WIQA-PL language constructs and explains how the language is used to express filtering policies.

Chapter 10: Explaining Assessment Results. This chapter describes the capabilities of the WIQA framework to explain filtering decisions.

Chapter 11: Implementation. This chapter describes the implementation of the WIQA framework and explains how applications interact with the framework.

Chapter 12: The WIQA Browser is an example application that uses the WIQA framework. The browser demonstrates how information quality filtering capabilities can be integrated into a standard Web browser. The browser enables users to extract RDF data from Web pages. The extracted data is stored together with provenance information in a local repository. The content of the local repository can be filtered using WIQA-PL policies. The user may retrieve explanations why displayed information satisfies the selected policy.

Chapter 13: Related Work. This chapter compares the WIQA Framework with related approaches.

Chapter 9

Expressing Information Filtering Policies

In general, a policy can be seen as a set of declarative rules which governs the behavior of an information system [CLW02, FWS05]. Instead of hard-coding system behavior at design time, policies allow to dynamically alter system behavior at run-time. Policy-based approaches to system management have been applied in various application domains such as network management, authentication, access control, privacy, digital rights management, and quality of service assurance [FWS05, Pie04]. Policies are usually expressed using a declarative policy language. Examples of standardized policy languages are the eXtensible Access Control Markup Language (XACML) [Mos05], an OASIS standard for defining access control policies, and the Platform for Privacy Preferences (P3P) [Mar02], a W3C standard for expressing privacy policies.

WIQA information filtering policies define which information is filtered positive by the WIQA filtering engine. WIQA policies are expressed using the WIQA-PL policy language. This chapter introduces the WIQA-PL language constructs and explains how the language is used to express filtering policies.

9.1 Basic Grammar

Figure 9.1 shows the Extended Backus-Naur Form (EBNF) [ISO96] definition for the basic grammar of the WIQA-PL policy language. The complete WIQA-PL grammar is given in appendix C.

Policies can be grouped into policy suites. A policy suite consists of a block of namespace prefix declarations and a set of policies. The namespace prefixes may be used later in PATTERN clauses to abbreviate URI references

```
1.  PolicySuite          ::= PrefixDeclaration*
2.                           Policy+
3.  PrefixDeclaration    ::= 'PREFIX' PrefixID Uriref
4.  Policy               ::= PolicyName
5.                           PolicyDescription?
6.                           PolicyPattern
7.                           RDFExplanationClause
8.  PolicyName           ::= 'NAME' Literal
9.  PolicyDescription    ::= 'DESCRIPTION' Literal
10. PolicyPattern        ::= 'PATTERN' PatternSet
```

Figure 9.1: EBNF grammar for the basic structure of a WIQA policy suite.

using the QName abbreviation mechanism [BHL06]. The PREFIX keyword associates a prefix label with a URI. A QName is mapped into an URI reference by concatenating its local part to the URI corresponding to its prefix.

```
1.  PatternSet           ::= '{' ExplanationClause?
2.                           GraphPattern*
3.                           FilterClause* '}'
4.  FilterClause         ::= 'FILTER' FilterExpression '.'
5.  ExplanationClause    ::= 'EXPL' ExplanationTemplates '.'
7.  GraphPattern         ::= GraphName '{'
8.                           ExplanationClause?
9.                           TriplePattern+
10.                          FilterClause* '}'
11. GraphName            ::= 'GRAPH' VariableOrUriOrANY
12. TriplePattern        ::= URIOrBnodeOrVariableOrReference
13.                          URIOrVariableOrReference
14.                          URIOrBnodeOrLiteralOrVariableOrReference '.'
15. VariableOrUriOrANY   ::= Variable | URI | 'ANY'
16. URIOrBnodeOrVariableOrReference
17.                      ::= URI | Bnode | Variable | Reference
18. URIOrVariableOrReference
19.                      ::= URI | Variable | Reference
20. URIOrBnodeOrLiteralOrVariableOrReference
21.                      ::= URI | Bnode | Literal | Variable |
22.                          Reference
23. Variable             ::= '?' String
24. Reference            ::= '?GRAPH' | '?SUBJ' |'?PRED' | '?OBJ'
```

Figure 9.2: EBNF grammar of the PATTERN clause.

Each policy consists of a NAME, a DESCRIPTION, and a PATTERN clause. The NAME clause specifies a display name for the policy. The

DESCRIPTION clause specifies a description for the policy that contains details about the quality indicators and assessment metrics that are used by the policy. The PATTERN clause specifies a set of conditions that triples have to satisfy in order to be filtered positive. The grammar of the PATTERN clause is shown in Figure 9.2. The grammar is based on the grammar of the SPARQL query language [PS05] in order to make it easier for people who already know SPARQL to learn WIQA-PL. A PATTERN clause may contain:

Graph Patterns which refer to the triples of the graph set to be filtered using a set of special, referring variables. The WIQA filtering engine matches the graph patterns against the set of named graphs to be filtered. The set of accepted triples is constructed from the matching results afterward. Chapter 9.2 describes how graph patterns are matched against graph sets. Chapter 9.3 describes how referring variables are used to link graph patters to the triples to be filtered, and how the set of accepted triples is constructed from the matching solutions.

Filter Clauses may be used to further restrict matching solutions. Filter clauses contain boolean-valued expressions consisting of variables, RDF terms, comparison operators, and function calls. Multiple expressions may be combined using logical operators. Chapter 9.5 explains filter clauses in detail.

Functions Calls. Quality-based information filtering policies may involve complex rating algorithms and statistical calculations. The WIQA filtering engine provides an extension mechanism for including arbitrary, application domain specific assessment functions into policies. Chapter 9.6 explains how function calls are used within filter clauses and describes the extension functions that have been implemented for the WIQA framework so far.

Explanation Clauses. Graph patterns may contain explanation clauses which define explanation templates. The templates consist of text fragments and variables. When a user requests an explanation why a triple satisfies a given policy, these templates are instantiated with variable bindings from the matching solutions. Chapter 10 describes the generation of explanations.

9.2 Graph Pattern Matching

A graph pattern consists of a *graph name pattern* and a set of *triple patterns*. The graph name pattern may either consist of a URI reference, a variable, or the keyword ANY. The graph name pattern ANY matches all graph names. Triple patterns consist of a subject, a predicate, and an object. The subject of a triple pattern may contain a URI, a bNode, or a variable. The predicate of a triple pattern has to be URI. The object of a triple pattern may contain a URI, a bNode, a literal, or a variable.

The variables contained in graph patterns are bound to RDF terms by matching the graph patterns against a set of named graphs. Let NG be a set of named graphs and GP a set of graph patterns. Let V be the set of all variables contained in GP, let RT be the set of all RDF terms contained in NG, and let GN be the set of all graph name URIs in NG. A matching solution s assigns an RDF term from RT to each variable in V. GP matches NG with the matching solution s if each variable in GP may be substituted with its value from s and if the keyword ANY may be substituted with a graph name from GN so that each graph pattern in GP is equal to or is a subgraph of a graph in NG.

Matching a set of graph patterns against a set of named graphs results in a *solution set* as multiple solutions may satisfy the condition above. This set is empty if no solution fulfills the condition above.

Figure 9.3 shows an example graph pattern. The WIQA-PL syntax for graph patterns requires each pattern to be introduced with the keyword GRAPH. The set of triple patterns is enclosed with curly brackets. Variable names are prefixed with a question mark. The graph pattern shown in Figure 9.3 consists of a graph name pattern and one triple pattern. The graph name pattern requires the matching graph to be named `fd:BackgroundInformation`. The triple pattern matches all triples with the predicate `foaf:name`. The subjects and objects of these triples are bound to the variables `?var1` and `?var2`. Matching the graph pattern against the example graph set from Chapter 7.2 results in the solution set shown in Table 9.1.

```
1. GRAPH fd:BackgroundInformation
2.   { ?var1 foaf:name ?var2 . }
```

Figure 9.3: Graph pattern 1.

Figure 9.4 shows another example graph pattern. The graph pattern consist of the graph name pattern ANY and two triple patterns. The graph pattern matches all graphs, independent of their graph name, that contain a triple having a `foaf:name` predicate and a triple having a `fin:country` predicate and an `iso:DE` object. Both triple patterns contain the variable `?var1` as subject. Therefore pairs of matching triples have to share the same subject.

	?var1	*?var2*
1.	`<mailto:peterSmith@deutsche-bank.de>`	`"Peter Smith" dtype:string`
2.	`<urn:x-DUNS:332907323>`	`"Deutsche Bank" dtype:string`
3.	`<mailto:reynolds@ft.com>`	`"John Reynolds" dtype:string`
4.	`<urn:x-DUNS:42307553>`	`"Financial Times" dtype:string`
5.	`<mailto:mark@scott.com>`	`"Mark Scott" dtype:string`

Table 9.1: Solution set from matching graph pattern 1 against the example graph set.

Matching the graph pattern against the example graph set from Chapter 7.2 results in the solution set shown in Table 9.2.

```
1. GRAPH ANY
2.   { ?var1 foaf:name ?var2 .
3.     ?var1 fin:country iso:DE . }
```

Figure 9.4: Graph pattern 2.

	?var1	*?var2*
1.	`<mailto:peterSmith@deutsche-bank.de>`	`"Peter Smith" xsd:string`
2.	`<urn:x-DUNS:332907323>`	`"Deutsche Bank" xsd:string`

Table 9.2: Solution set from matching graph pattern 2 against the example graph set.

Figure 9.5 shows a graph pattern set consisting of two graph patterns. The pattern set demonstrates how the condition, that information should originate from John Reynolds, is expressed by combining two graph patterns. The first graph pattern consists only of variables and therefore matches every triple. The second graph pattern matches only triples in the graph `fd:GraphFromAggregator` that use the Semantic Web Publishing Vocabulary to describe the origin of the graphs that have been asserted by Dave Reynolds. The names of these graphs are bound to the variable `?graph`. When both patterns are matched simultaneously against a set of named graphs, the second pattern works like an additional filter for the first pattern: The second pattern binds the names of all graphs that have been asserted by John Reynolds to the variable `?graph`. With this constraint for the variable `?graph`, the first pattern matches only triples within graphs from John Reynolds and the subjects, predicates, and objects of these triples are bound to the variables `?var1`,

?var2, and ?var3. Matching both graph patterns together against the example graph set from Chapter 7.2 results in the solution set shown in Table 9.3.

```
GRAPH ?graph
  { ?var1 ?var2 ?var3 . }

GRAPH fd:GraphFromAggregator
  { ?graph   swp:assertedBy ?warrant .
    ?warrant swp:authority <mailto:reynolds@ft.com> . }
```

Figure 9.5: Graph pattern set.

Variable	*Solution 1*	*Solution 2*
?var1	`<urn:x-DUNS:316067164>`	`<urn:x-DUNS:047897855>`
?var2	`fin:news`	`fin:news`
?var3	`"Siemens AG ..."@EN`	`"Intel has ..."@EN`
?graph	`fd:GraphFromJohnReynolds`	`fd:GraphFromJohnReynolds`
?warrant	`fd:JrWarrant`	`fd:JrWarrant`

Table 9.3: Solution set from matching the graph pattern set from Figure 9.5 against the example graph set.

9.3 Accepting Triples

The WIQA-PL policy language uses graph patterns to represent conditions that triples have to satisfy in order to be filtered positive. When a policy is applied, the WIQA filtering engine checks for each triple in the graph set to be filtered whether it satisfies the conditions given in the pattern clause of the current policy. The triples that satisfy the conditions are included into the set of accepted triples.

Conditions are expressed as graph patterns which refer to the triples in the graph set to be filtered using a set of special variables. These *referring variables* connect the graph patterns in the pattern clause with the triples in the graph set to be filtered. The referring variables are shown in Table 9.4. If the variable ?SUBJ is used in any graph pattern, then only triples with a subject that equals a binding of the variable ?SUBJ are accepted. If the variable ?PRED is used in any pattern, then only triples are filtered positive which have a predicate matching a binding of the variable ?PRED. If the variable ?OBJ

Variable	*Description*
?SUBJ	Reference to subject of a triple.
?PRED	Reference to predicate of a triple.
?OBJ	Reference to object of a triple.
?GRAPH	Reference to the graph containing a triple.

Table 9.4: WIQA-PL referring variables.

is used, then only triples are filtered positive with an object matching a `?OBJ` value. If the variable `?GRAPH` is used in any pattern, then only those triples are accepted that occur in a graph that is named with an URI reference that equals a binding of the variable `?GRAPH`. If multiple referring variables are used within the same pattern clause then triples are accepted only if they match values of all referring variables and these values occur in a single matching solution.

When a policy is applied against a graph set, the WIQA engine generates the set of accepted triples by conducting the following steps:

1. The filtering engine adds the graph pattern `GRAPH ?GRAPH { ?SUBJ, ?PRED, ?OBJ }` to the set of graph patterns given by the pattern clause of the policy. In the following, this graph pattern will be called *root pattern*.

2. The engine matches the extended pattern set against the graph set to be filtered. This results into a solution set.

3. The engine generates an accepted triple from each distinct set of values of the variables `?SUBJ, ?PRED and ?OBJ` in the solution set.

Figure 9.6 shows the WIQA-PL representation of the policy "Accept only information which has been asserted by German analysts". Lines 1-5 define the namespace prefixes which are used in the pattern clause later. Line 7 and 8 specify the policy name and policy description. The pattern clause restricts information to originate from German analysts. It consists of two graph patterns. The first graph pattern requires provenance information about graphs to be contained in the graph `fd:GraphFromAggregator`. It contains two triple patterns which require provenance information to be expressed using the SWP properties `swp:assertedBy` and `swp:authority`. The first pattern binds the names of asserted graphs to the referring variable `?GRAPH`. The second pattern binds URIs that identify authorities to the variable `?authority`. The second triple pattern is connected with the first one by sharing the variable `?warrant`.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix swp: <http://www.w3.org/2004/03/trix/swp-2/> .
@prefix iso: <http://www.daml.org/2001/09/countries/iso-3166-ont#> .
@prefix fin: <http://www.fu-berlin/suhl/bizer/2006/FinVoc/> .
@prefix fd: <http://www.fu-berlin/suhl/bizer/exampleDataset#> .

NAME "Information from German analysts"
DESCRIPTION "Use only information which has been asserted by
             German analysts."
PATTERN
     {
       GRAPH fd:GraphFromAggregator
         { ?GRAPH swp:assertedBy ?warrant .
           ?warrant swp:authority ?authority . }

       GRAPH fd:BackgroundInformation
         { ?authority rdf:type fin:Analyst .
           ?authority fin:country iso:DE . }
     }
```

Figure 9.6: WIQA-PL policy: Use only information which has been asserted by German analysts.

The second graph pattern requires authorities to be an instance of the class `fin:Analyst` and to have a `fin:country` property with the value `iso:DE`. The triples that describe authorities have to occur in the graph `fd:BackgroundInformation`.

When the policy is applied against the example graph set from Section 7.2, the second graph pattern matchs the analyst Peter Smith and the variable `?authority` is bound to `<mailto:peterSmith@deutsche-bank.de>`. With this binding of the variable `?authority`, the first graph pattern matches the graph `fd:GraphFromAggregator` and binds the value `fd:GraphFromPeterSmith` to the referring variable `?GRAPH`. With this binding of the variable `?GRAPH`, the WIQA engine filters all triples that are contained in `fd:GraphFromPeterSmith` positive. The engine would therefore return the set of accepted triples shown in Figure 9.7 to the application.

9.4 Context Variables

Filtering policies may rely on information about the application context in which they are applied. Subjective policies might, for instance, require information about the user who applies them; time-dependent policies might

```
<urn:x-ISIN:DE0007236101> fin:positiveAnalystReport "As Siemens
  agrees partnership with Novell unit SUSE ..."@EN .
<urn:x-ISIN:US4581401001> fin:negativeAnalystReport "Chiphersteller
  Intel will nach Firmenangaben mit milliardenschweren ..."@DE .
```

Figure 9.7: Accepted triples.

require the current time.

Applications can provide the WIQA engine with information about the application context by setting context variables at run-time. Context variables can be used within WIQA-PL policies. The names of context variables are written in uppercase letters in order to distinguish them from other variables. Before applying a policy, the WIQA engine substitutes all context variables within the policy with their values set by the application. As the variable names ?GRAPH, ?SUBJ, ?PRED, ?OBJ are already reserved for the referring variables, these names cannot be used for context variables.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix swp: <http://www.w3.org/2004/03/trix/swp-2/> .
@prefix iso: <http://www.daml.org/2001/09/countries/iso-3166-ont#> .
@prefix fin: <http://www.fu-berlin/suhl/bizer/2006/FinVoc/> .
@prefix fd: <http://www.fu-berlin/suhl/bizer/exampleDataset#> .

NAME "Information from positively rated information providers"
DESCRIPTION "Use only information from information providers that
             I have rated positive."
PATTERN
     {
       GRAPH fd:GraphFromAggregator
         { ?GRAPH swp:assertedBy ?warrant .
           ?warrant swp:authority ?authority . }

       GRAPH ?myGraph
         { ?USER fin:positiveRating ?authority . }

       GRAPH fd:GraphFromAggregator
         { ?myGraph swp:assertedBy ?warrant2 .
           ?warrant2 swp:authority ?USER . }
     }
```

Figure 9.8: WIQA-PL policy: Use only information from information providers that I have rated positive.

Figure 9.8 shows the WIQA-PL representation of the policy “Use only information from information providers that I have rated positive”. In order to determine which information providers have been rated positive by the current user, the WIQA engine requires the URI reference identifying the current user. This URI reference is represented by the context variable `?USER` within the policy.

The PATTERN clause of the policy contains three graph patterns. The first pattern binds the URI references identifying information providers to the variable `?authority`. The second graph pattern restricts `?authority` bindings to information providers that have been rated positive by the current user. The names of the graphs that contain the ratings are bound to the variable `?myGraph`. The third graph pattern ensures that ratings originate from the current user by requiring the graph `fd:GraphFromAggregator` to contain the statements that `?myGraph` was asserted by the current user.

9.5 Filters

FILTER clauses restrict solution sets according to a given expression. They eliminate any solution from the solution set that, when substituted into the expression, results in a boolean value of false or produces an error.

Figure 9.9 shows the grammar of the FILTER clause. Filter expressions may consist of relational, logical, and numeric expressions and may include function calls:

Relational Expressions compare variables, RDF terms, and numeric expressions to each other. Relational expressions may use the following comparison operators: = (equal), != (not equal), > (greater than), < (less than), >= (greater than or equal to) and <= (less than or equal to). The evaluation of a relational expressions results either in the boolean value true or false. WIQA-PL uses the same rules for comparing RDF terms as the SPARQL query language. These rules are defined in section 11.3 of the SPARQL specification [PS05]. An example of a relational expression is `?authority = <mailto:chris@bizer.de>`, meaning that the value of the variable `?authority` has to be the URI `<mailto:chris@bizer.de>`.

Logical Expressions connect multiple expressions using the && (logical AND) and || (logical OR) operators or negate the result of an expression using the ! (logical NOT) operator. An example of a logical expression using the && operator is `?authority != <mailto:chris@bizer.de> && ?price < 20`, meaning that the value of the variable `?authority` has to

```
1.  FilterClause        ::= 'FILTER' Expression | FunctionCall
2.  Expression          ::= '(' ConditionalOrExpression ')'
3.  ConditionalOrExpression ::= ConditionalAndExpression ( '||'
4.                          ConditionalAndExpression )*
5. ConditionalAndExpression ::= LogicalValue ( '&&' LogicalValue )*
6. LogicalValue         ::= RelationalExpression
7. RelationalExpression ::= NumericExpression ( '=' NumericExpression |
8.                          '!=' NumericExpression |
9.                          '<'  NumericExpression  |
10.                         '>'  NumericExpression |
11.                         '<=' NumericExpression |
12.                         '>=' NumericExpression )?
13. NumericExpression  ::= AdditiveExpression
14. AdditiveExpression ::= MultiplicativeExpression (
15.                         '+' MultiplicativeExpression |
16.                         '-' MultiplicativeExpression )*
17. MultiplicativeExpression ::= UnaryExpression ( '*' UnaryExpression |
18.                         '/' UnaryExpression )*
19. UnaryExpression    ::= '!' PrimaryExpression |
20.                         '+' PrimaryExpression |
21.                         '-' PrimaryExpression |
22.                         PrimaryExpression
23. PrimaryExpression  ::= Expression | FunctionCall | URIref |
24.                         RDFLiteral | NumericLiteral | BooleanLiteral
25.                         | BlankNode | Variable
```

Figure 9.9: Grammar of the FILTER clause.

be different from the URI `<mailto:chis@bizer.de>` and the value of the variable `?price` has to be smaller than 20.

Numeric Expressions conduct calculations. Numeric expressions may use the + (add), - (substract), * (multiply), and / (divide) operators. WIQA-PL uses the same functions for evaluating numeric expressions as the SPARQL query language. These functions are defined in section 11.3 of the SPARQL specification [PS05]. An example of a relational expression that includes a numeric expression is `?priceYen < ?priceDollar * 112`.

Figure 9.10 shows the WIQA-PL policy "Accept only information that has been asserted after January 1st, 2006 by analysts who achieved a StarMine score above 80". The policy uses two filter clauses: The filter clause in line 10 restricts bindings of the variable `?date` to values that are greater than 2006-01-01. The FILTER clause in line 19 restricts bindings of the variable `?benchmark` to integer values greater than 80.

```
NAME "New information from highly rated analysts"
DESCRIPTION "Accept only information that has been asserted
             after January 1st, 2006 by analysts who achieved
             a StarMine score above 80."
PATTERN
      {
       GRAPH fd:GraphFromAggregator
          { ?GRAPH swp:assertedBy ?warrant .
            ?warrant swp:authority ?authority .
            ?warrant dc:date ?date .
            FILTER (?date > "2006-01-01"^^xsd:date) . }

       GRAPH fd:BackgroundInformation
         { ?authority rdf:type fin:Analyst .
           ?authority fin:benchmark ?benchmark .
           FILTER (?benchmark > "80"^^xsd:integer) . }
      }
```

Figure 9.10: WIQA-PL policy: Accept only information that has been asserted after January 1st, 2006 by analysts who achieved a StarMine score above 80.

9.6 Functions

```
FunctionCall ::= RDFrelatedFunction | CastingOrExtensionFunction
CastingOrExtensionFunction  ::=  URIref ArgList
ArgList ::= ( '(' NIL | Expression ( ',' Expression )* ')' )
RDFrelatedFunction ::= 'str' '(' Expression ')' |
                       'lang' '(' Expression ')' |
                       'datatype' '(' Expression ')' |
                       'isUri' '(' Expression ')' |
                       'isBlank' '(' Expression ')' |
                       'isLiteral' '(' Expression ')' |
                       'regex' '(' Expression ',' Expression (
                       ',' Expression )? ')'
```

Figure 9.11: Grammar for WIQA-PL function calls.

Filter expressions may include function calls. A function takes some number of RDF terms as arguments and returns an RDF term or a boolean value as result. Figure 9.11 shows the grammar for invoking functions within filter clauses. There are three types of functions in WIQA-PL:

Basic RDF-Related Functions are used to test RDF-specific properties of variable bindings. RDF-related functions can, for instance, be used to check if a variable is bound to a URI reference or to a literal, or to test if a literal has a specific language tag. WIQA-PL provides the same RDF-related functions as the SPARQL query language. Table 9.5 contains a short description of each function. The functions are specified in detail in chapter 11.4 of the SPARQL specification [PS05]. The policy suite that is shown in Figure 9.12 contains two policies that use RDF-related functions: The policy "Accept only German or English information" uses the `lang()` function to check whether the language tag of RDF literals has the value DE or EN. The policy "Accept only information from Deutsche Bank" uses the the `str()` and the `regex()` functions to check whether the URI that identifies an authority contains the domain name `deutsche-bank.de`.

Constructor Functions are used to cast literals to a specific datatype. Casting is performed by calling a constructor function for the target type on an operand of the source type. WIQA-PL provides the same constructor functions as the SPARQL query language. Table 9.6 gives an overview of these functions. The functions are defined in detail in chapter 11.5 of the SPARQL specification [PS05]. Each constructor function can cast only a specific set of datatypes into the target type. For instance, a `xsd:float` cannot be casted into a `xsd:dateTime`. The datatypes that are allowed for the operand of each constructor function are also specified in chapter 11.5 of the SPARQL specification [PS05]. Calling a constructor function with an operand that has a disallowed datatype raises an error. Constructor functions are used within filter clauses to make literals that have different datatypes comparable. For instance, the filter clause `FILTER (xsd:dateTime(?date) > "2005-11-20T17:22:10"^^xsd:dateTime)` uses the constructor function `xsd:dateTime()` to cast the value of the variable `?date` to the datatype `xsd:dateTime` before comparing it to the given value.

Extension Functions. WIQA-PL provides an extension mechanism for invoking arbitrary, application domain specific functions. Extension functions are implemented as plug-ins for the WIQA filtering engine. An extension function is named by a URI and takes some number of RDF terms as arguments. The result of an extension function is an RDF term. Extension functions are called within policies by their URI followed by a list of arguments. The list of arguments is enclosed with parentheses and arguments are separated by commas.

As we have seen in Chapter 3, quality-based information filtering policies rely on a wide range of different, application domain specific assessment

Function	*Description*
isUri()	Returns true if the argument is a URI. Returns false otherwise.
isBlank()	Returns true if the argument is a blank node.
isLiteral()	Returns true if the argument is a literal.
lang()	Returns the language tag of a literal, if it has one.
datatype()	Returns the datatype URI of a literal.
str()	Returns an string representation of a URI reference.
regex()	Invokes the Xpath [CD99] regular expression function to match a string against a regular expression.

Table 9.5: Basic RDF-related functions.

```
NAME "Only German or English information"
DESCRIPTION "Accept only German or English information.
      The language is determined by testing the RDF language tag."
PATTERN
      { FILTER( lang(?OBJ) = 'DE' || lang(?OBJ) = 'EN' ) . }

NAME "Accept only information from Deutsche Bank"
DESCRIPTION "Checks if information has been asserted by an
             authority identified with a email address within
             the domain 'deutsche-bank.de'."
PATTERN {
      GRAPH ANY {
        ?GRAPH swp:assertedBy ?warrant .
        ?warrant swp:assertedBy ?authority .
        FILTER(regex(str(?authority), 'deutsche-bank\.de')) . } }
```

Figure 9.12: WIQA-PL policy suite containing two policies using WIQA-PL build in functions.

metrics. For instance, rating-based filtering policies use various scoring algorithms to calculate the score for an entity from a network of ratings. Content-based filtering policies may rely on natural language processing methods to analyze text or may use various statistical methods to compare a piece of information with related information. By including domain specific functions, the WIQA framework can be extended to fit the requirements of different application domains.

Three example extension functions have been implemented so far: The More Positive Ratings and the Tidal Trust functions implement different rating-based scoring algorithms; the `wiqa:count` function allows the formulation of quantity constraints. The functions are named with URIs in the

Function	*Description*
xsd:boolean()	Produces a typed literal with the datatype `xsd:boolean` from the operand.
xsd:double()	Produces a literal with the datatype `xsd:double`.
xsd:float()	Produces a literal with the datatype `xsd:float`.
xsd:decimal()	Produces a literal with the datatype `xsd:decimal`.
xsd:integer()	Produces a literal with the datatype `xsd:integer`.
xsd:dateTime()	Produces a literal with the datatype `xsd:dateTime`.
xsd:string()	Produces a literal with the datatype `xsd:string`.

Table 9.6: Constructor functions.

namespace `http://www.wiwiss.fu-berlin.de/suhl/bizer/WIQA/`, which is abbreviated using the `wiqa:` prefix. The extension functions will be described in the following sections.

9.6.1 More Positive Ratings Function

The More Positve Ratings extension function implements a simple rating-based scoring algorithm. The function counts all positive and negative ratings for a resource within the graph set to be filtered. It returns true, if the resource received more positive than negative ratings, and returns false otherwise.

The function assumes that ratings are expressed using the terms `fin:positiveRating` and `fin:negativeRating` from the financial vocabulary introduced in Section 7.2. The function has one operand that determines the resource for which the ratings are counted.

Figure 9.13 shows the WIQA-PL policy “Only accept information from information providers who have received more positive than negative ratings”. The filter clause in line 8 uses the `wiqa:MorePositiveRatings` function to check whether an `?authority` has received more positive than negative ratings.

An advantage of the `wiqa:MorePositiveRatings()` function is that the evaluation process is easy to understand for the information consumer. A disadvantage of the function is that it is very susceptible to ballot stuffing and bad mouthing attacks (see Section 3.1.3), as it takes all ratings into account and does not differentiate between ratings from trustworthy and less trustworthy raters.

```
NAME "More positive Ratings"
DESCRIPTION "Only accept information from information providers who
             have received more positive than negative ratings."
PATTERN
      {  GRAPH fd:GraphFromAggregator
              { ?GRAPH swp:assertedBy ?warrant .
                ?warrant swp:authority ?authority .
                FILTER wiqa:MorePositiveRatings(?authority) . }
     }
```

Figure 9.13: WIQA-PL policy: Only accept information from information providers who have received more positive than negative ratings.

9.6.2 Tidal Trust Function

The Tidal Trust extension function implements a more complex rating-based scoring algorithm. The Tidal Trust algorithm was developed by Jennifer Golbeck at the University of Maryland [Jen05]. The algorithm takes only ratings from information providers into account who are on the information consumer's web-of-trust. Ratings from other information providers are ignored. The ratings are weighted with the degree of trust the information consumer has in the information provider. The Tidal Trust algorithm is therefore more robust against ballot stuffing and bad mouthing attacks than the algorithm described in the last section. By weighting the ratings, the algorithm can take the personal bias of the information consumer into account and is therefore suitable for situations where ratings are subjective [GM06].

The algorithm operates on a network of ratings in which each node has rated several other nodes. The meaning of the ratings may differ between application scenarios. Within the financial information integration scenario from Chapter 7, a rating may, for instance, represent an investor's opinion about the quality of discussion forum postings from another investor. Figure 9.14 shows an example rating network. The nodes represent information providers, the edges represent ratings on a scale from 1 (low quality) to 10 (high quality).

The Tidal Trust algorithm determines the rating of a node in the network (called *sink*) from the perspective of another node (called *source*). If the rating network contains a direct rating by the source for the sink, then the algorithm returns this rating as result. If the network does not contain such a rating, the algorithm infers an approximated rating from the paths connecting the two nodes.

The inference is based on two assumptions: It is expected that people who the user rates highly will tend to agree with the user more about the

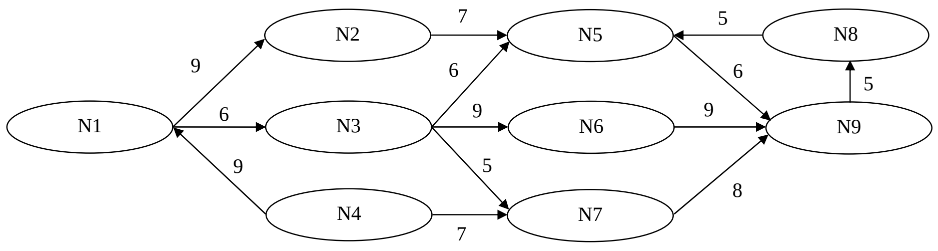

Figure 9.14: Example rating network.

rating of others than people whom the user gives a low rating. Second, the accuracy of inferred ratings is expected to decrease with length of the paths that connect two individuals. These assumptions are motivated in [GM06] with the experience from several real-world rating networks.

Based on these assumptions, the Tidal Trust algorithm infers a missing rating by conducting the following steps:

1. It searches for all minimum length paths in the network that connect the source with the sink. The length of a path is understood as the number of edges that form a path. Let's assume, for example, that node N1 wants to infer a rating for node N9 from the example network shown in Figure 9.14. There are four minimal length paths from N1 to N9: The first path connects the source with the sink over the nodes N2 and N5; the second path over the nodes N3 and N5; the third path over the nodes N3 and N6 and the forth path over nodes N3 and N7.

2. The algorithm determines the *strength* of each path. The strength of a path equals the lowest rating on the path. Within our example, the strength of paths 1-3 is 6, the strength of the fourth path is 5.

3. Afterwards, the algorithm establishes the threshold *max*. This threshold is used to determine which ratings are taken into account in the final calculation. The threshold *max* is set to the maximum strength of all minimal length paths leading to the sink. Within our example, *max* has the value 6.

4. With the *max* value established, each node on a path which does not have a rating for the sink can calculate the rating as the weighted average of the ratings for the sink from its successors using the formula shown in Figure 9.15. t_{ij} stands for the rating of a node i for node j. The function $suc(i)$ returns all nodes that are successors of node i. The formula takes only ratings from nodes into account which are rated at or

$$t_{is} = \frac{\sum_{j \in suc(i) \mid t_{ij} \geq max} t_{ij}\, t_{js}}{\sum_{j \in suc(i) \mid t_{ij} \geq max} t_{ij}}$$

Figure 9.15: Formula for inferring the rating for a sink node s from the perspective of a source node i.

above the max threshold by their predecessor. Each rating is weighted by the rating a successor received from its predecessor. Within our example network, nodes N5, N6 and N7 already have ratings for the sink N9. Node N3 infers its rating for the sink from the ratings of nodes N5 and N6. It does not take the rating from node N7 into account as its rating for node N7 is below the max threshold of 6. Formula 9.15 results in the value 7.8 for the ratings of N5 and N6 and the ratings of N3 for both nodes. Node N2 infers the rating 6 from the rating of node N3. The source node N1 infers a rating of 6.72 for the sink from the ratings of nodes N2 and N3.

The WIQA implementation of the Tidal Trust algorithm assumes that ratings are represented using the FOAF Trust Module [Jen05]. Ratings have to be contained in the graph set to be filtered. The extension function has two arguments: The first argument identifies the source node; the second argument identifies the sink node. Figure 9.16 shows a WIQA-PL policy that uses the Tidal Trust extension function. The policy accepts information only from information providers who have a Tidal Trust rating that is greater than 5.

```
NAME "TidalTrust rating above 5"
DESCRIPTION "Only accept information from information providers with
             a Tidal Trust rating above 5."
PATTERN
      {  GRAPH fd:GraphFromAggregator
               { ?GRAPH swp:assertedBy ?warrant .
                 ?warrant swp:authority ?authority .
                 FILTER (wiqa:TidalTrust(?USER, ?authority) > 5) }
      }
```

Figure 9.16: WIQA-PL policy: Only accept information that originates from information providers with TidalTrust rating above 5.

9.6.3 Count Function

Filtering policies may rely on quantity constraints. For example, a policy might require information to be asserted by a number of independent information sources. Other policies might require information sources to have received a certain number of positive ratings; or might accept information only from information providers that are believed to be experts on a specific topic because they have worked for a certain number of projects involving that topic.

The WIQA extension function `wiqa:count()` is used to express quantity constraints within WIQA policies. `wiqa:count()` takes one variable as operand and returns the number of different RDF terms that are bound to this variable within a group of matching solutions. The grouping is defined by the position of the filter clause containing the `wiqa:count()` function in the pattern clause. The filter clause can either be included into a graph pattern or it can be positioned after the graph patterns.

If the filter clause is positioned after the graph patterns, then the solution set is grouped by the variables `?SUBJ`, `?PRED`, and `?OBJ`. A group of solutions is formed by all solutions within the solution set that assign the same values to these variables.

```
1.  NAME "Asserted by two different analysts"
2. DESCRIPTION "Only accept information that has been asserted by
3.              at least two different analysts."
4. PATTERNS
5.      {
6.         GRAPH ANY { ?GRAPH swp:assertedBy ?warrant .
7.                     ?warrant swp:authority ?authority . }
8.
9.         GRAPH ANY { ?authority rdf:type fin:Analyst . }
10.
11.        FILTER (wiqa:count(?authority) >= 2) .
12.     }
```

Figure 9.17: WIQA-PL policy: Only accept information that has been asserted by at least two different analysts.

Figure 9.17 shows the policy “Only accept information that has been asserted by at least two different analysts”. The pattern clause of the policy consists of two graph patterns that are followed by a filter clause using the `wiqa:count()` function (line 11).

Figure 9.18 shows an example graph set consisting of three graphs. `ex:Graph3` contains provenance information about `ex:Graph1` and `ex:Graph2`

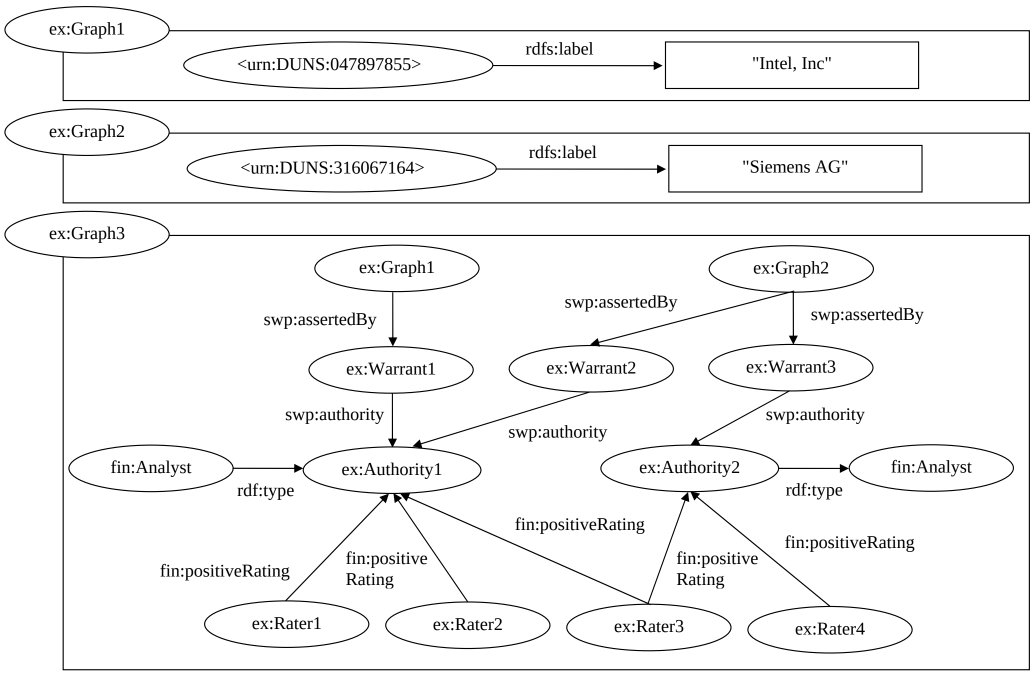

Figure 9.18: Example graph set.

and background information about the authorities that have asserted the graphs: `ex:Authority1` has asserted `ex:Graph1` and `ex:Graph2`. `ex:Authority1` is a `fin:Analyst` and has received three positive ratings. `ex:Authority2` has asserted `ex:Graph2`. `ex:Authority2` is also a `fin:Analyst` but has received only two positive ratings.

When the policy is applied against the example graph set, the WIQA filtering engine conducts the following steps:

1. The filtering engine adds the root pattern `GRAPH ?GRAPH { ?SUBJ, ?PRED, ?OBJ . }` to the set of graph patterns given by the pattern clause. Matching all three patterns against the example graph set results in the solution set shown in Table 9.7.

2. As the filter clause, that contains the `wiqa:count()` function, is positioned after the graph patterns, the solution set is grouped by the variables `?SUBJ`, `?PRED` and `?OBJ`. This results in two groups: The first group contains solution 1; the second group contains solution 2 and 3, as these solutions assign the same values to all three variables.

3. The operand of the `wiqa:count()` function is the variable `?authority`. The function therefore counts the number of different values of this

variable in each group. It returns the value 1 for solutions in the first group, and the value 2 for solutions in the second group.

4. The filter clause requires solutions to have a `wiqa:count()` result greater or equal to 2. Solution 1 is therefore removed from the solution set and the accepted triple `<urn:x-DUNS:316067164> rdfs:label "Siemens AG"` is constructed from the remaining second solution set.

Variable	*Solution 1*	*Solution 2*	*Solution 3*
?SUBJ	`<urn:x-DUNS:047897855>`	`<urn:x-DUNS:16067164>`	`<urn:x-DUNS:316067164>`
?PRED	`rdfs:label`	`rdfs:label`	`rdfs:label`
?OBJ	`"Intel, Inc"`	`"Siemens AG"`	`"Siemens AG"`
?GRAPH	`ex:Graph1`	`ex:Graph2`	`ex:Graph2`
?warrant	`ex:Warrant1`	`ex:Warrant2`	`ex:Warrant3`
?authority	`ex:Authority1`	`ex:Authority1`	`ex:Authority2`

Table 9.7: Solution set from matching all three patterns against the example graph set.

The `wiqa:count()` function may also be used within graph patterns. Figure 9.19 shows the policy “Only accept information that has been asserted by analysts who have received at least 3 positive ratings”. The policy consists of three graph patterns. The third graph pattern contains the filter clause `FILTER (wiqa:count(?rater) > 2)` (line 11).

```
1.  NAME "Asserted by analysts with at least 3 positive ratings."
2.  DESCRIPTION "Only accept information that has been asserted by
3.                  analysts who have received at least 3 positive ratings."
4.  PATTERNS {
5.           GRAPH ANY { ?GRAPH swp:assertedBy ?warrant .
6.                       ?warrant swp:authority ?authority . }
7.
8.           GRAPH ANY { ?authority rdf:type fin:Analyst . }
9.
10.          GRAPH ANY { ?rater fin:positiveRating ?authority .
11.                      FILTER (wiqa:count(?rater) > 2) . }
12.          }
```

Figure 9.19: WIQA-PL policy: Only accept information that has been asserted by analysts who have received at least 3 positive ratings.

The graph patterns in the pattern clause are connected to the root pattern by the referring variables `?GRAPH`, `?SUBJ`, `?PRED`, or `?OBJ`. The graph patterns may also share variables between each other. All variables that occur in more than on graph pattern are called *shared variables*. The first graph pattern (line 5-6) in Figure 9.19 refers to the root pattern by using the variable `?GRAPH`. All three graph patterns share the variable `?authority`.

Graph patterns form a pattern tree by sharing variables. Figure 9.20 shows the pattern tree for the policy shown in Figure 9.19.

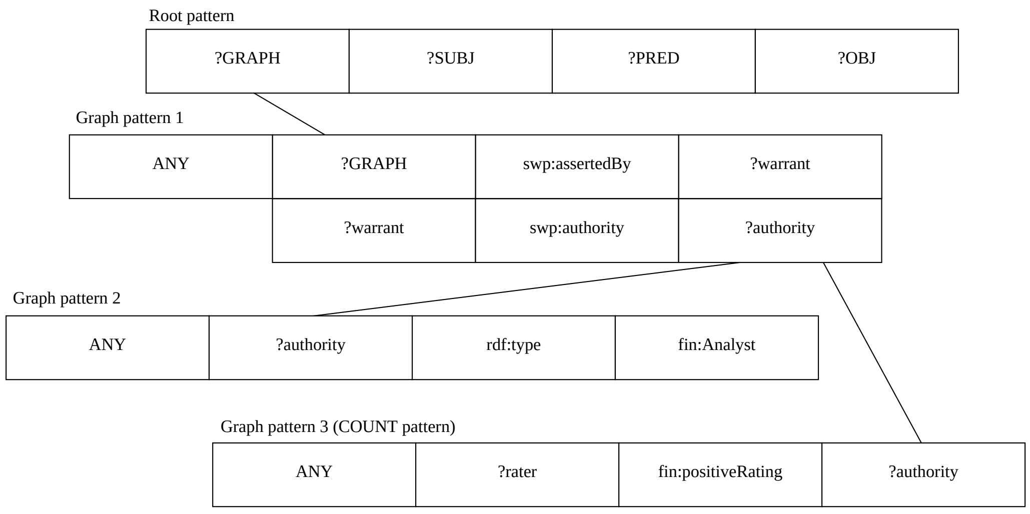

Figure 9.20: Graph pattern tree for the policy: Only accept information that has been asserted by analysts who have received at least 3 positive ratings.

When the `wiqa:count()` function is used within a graph pattern, then the solution set is grouped by the variables `?SUBJ`, `?PRED`, and `?OBJ` and by all shared variables that lie on the path between the root pattern and the graph pattern containing the `wiqa:count()` function.

For our example policy, the shortest path is formed by the variables `?GRAPH` and `?authority`. The solution set is therefore grouped by the variables `?SUBJ`, `?PRED`, `?OBJ`, `?GRAPH`, and `?authority`.

Applying the example policy against the graph set shown in Figure 9.18 leads to a solution grouping that consists of three groups: The first group contains all solutions resulting from the triple in `ex:Graph1`, its assertion by `ex:Authority1` and the three ratings of `ex:Authority1`. The second group contains all solutions resulting from the triple in `ex:Graph2`, its assertion by `ex:Authority1` and the three ratings of `ex:Authority1`. The third group contains all solutions resulting from the triple in `ex:Graph2`, its assertion by `ex:Authority2` and the two ratings of `ex:Authority2`.

The function call `wiqa:count(?rater)` returns the number of different values of the variable `?rater` within the group to which a solution belongs. The number is required to be greater than two by the filter clause. `ex:Authority2` was rated by two raters only, therefore all solutions from the third group are removed from the solution set. As `ex:Authority1` was rated by three different raters, the solutions from the first and second group remain in the solutions set, which finally leads to the acceptance of all triples from `ex:Graph1` and `ex:Graph2`.

9.7 Summary

This section introduced the WIQA-PL Policy Language and explained how the language is used to express different filtering policies. Within WIQA-PL, filtering policies are expressed as a set of conditions that a piece of information has to satisfy in order to be filtered positive. The language assumes that information is represented as a set of named graphs. Conditions are expressed as graph patterns which refer to the triples to be filtered using a set of referring variables. Graph patterns may contain filter clauses, which further restrict pattern matches. Filter clauses may consist of relational, logical, and numeric expressions and may include function calls. WIQA-PL policies can invoke application domain specific assessment metrics through an extension function mechanism.

The design of the language was lead by the following goals:

Flexibility. As we have seen in Chapter 3, quality-based information filtering policies combine a wide range of different context-, content-, and rating-based assessment metrics. A language for expressing policies should therefore provide a high degree of flexibility. WIQA-PL tries to achieve this flexibility by relying on constructs from RDF query languages, such as graph patterns and filters, which have already proven their general applicability.

Extensibility. Information quality assessment often requires domain-specific metrics. In order to be applicable across different domains, an information quality assessment framework should be extensible with domain-specific metrics. WIQA-PL provides this extensibility through its extension function mechanism.

Standard Conformance. Whenever possible, newly introduced languages should be based on well-known concepts. WIQA-PL adopts the concepts of graph patterns and filters and the syntax for representing them from SPARQL, the standard query language for RDF. This may make it easier for users who are already familiar with SPARQL to learn WIQA-PL.