

# SemWIK

# A Semantic Web Middleware for Virtual Data Integration on the Web

Andreas Langegger, Wolfram Wöß, Martin Blöchl – Institute for Applied Knowledge Processing, Johannes Kepler University Linz, Austria  
Contact: Andreas Langegger <al@jku.at>, <http://semwiq.faw.uni-linz.ac.at>

## Abstract

The *Semantic Web Integrator and Query Engine* (SemWIK) is a mediator which provides access to distributed, heterogeneous data sources applying Semantic Web technology. It allows to retrieve data using SPARQL queries, data sources can register and abandon freely, and all RDF Schema and OWL vocabularies can be used to describe data at

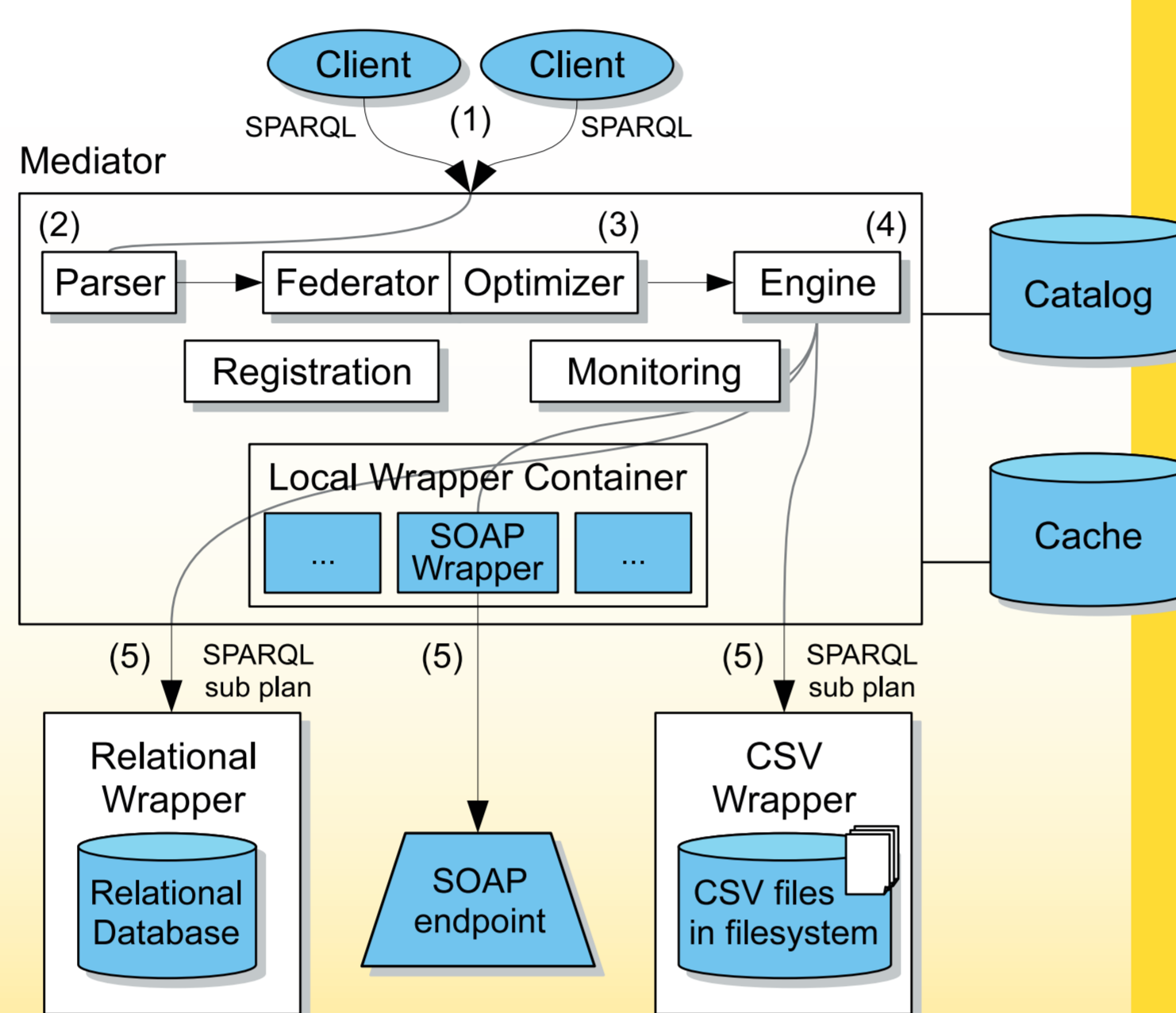
end-points. Data heterogeneity is addressed by RDF-wrappers like D2R-Server placed on top of local information systems. A query does not directly refer to actual end-points, instead it contains graph patterns adhering to a virtual dataset. The mediator retrieves and joins RDF data from different end-points providing a transparent view to the end-user.

## Concept & Architecture

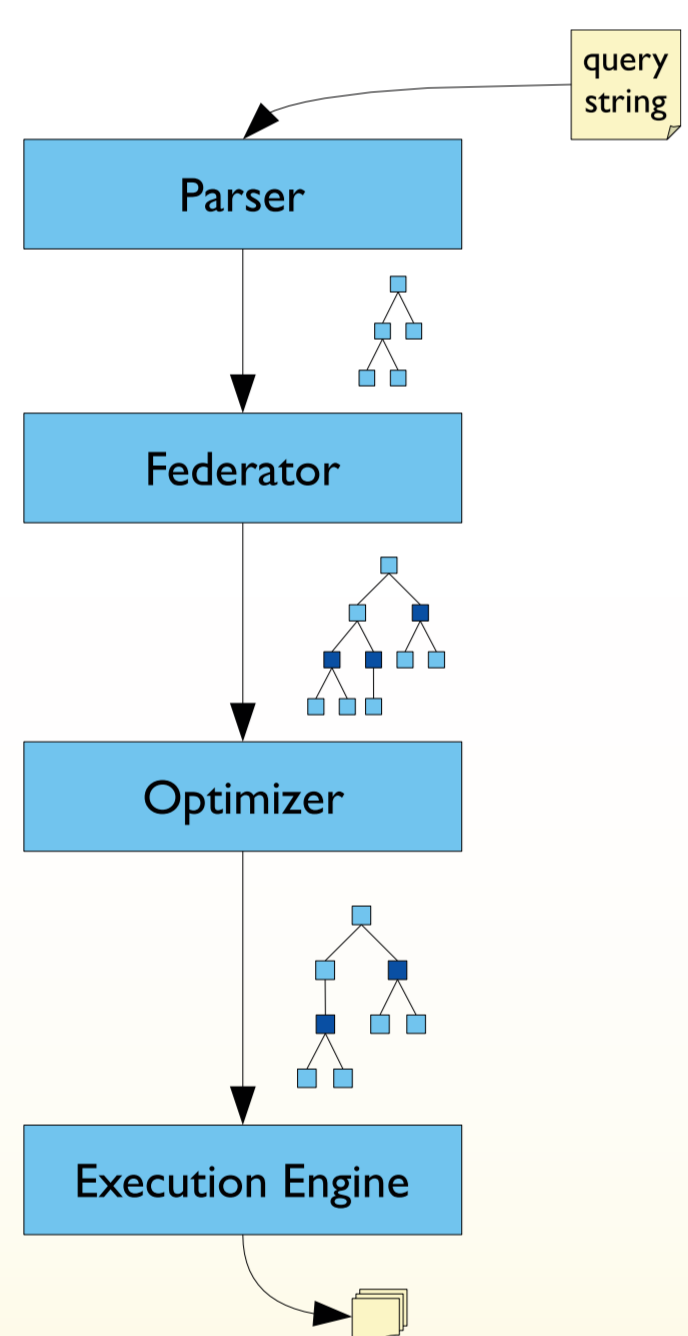
For the implementation of SemWIK the Jena RDF library is used. The figure on the bottom right depicts the typical mediator-wrapper architecture applied to SemWIK.

Clients establish a connection to the mediator and request data by submitting SPARQL queries (1). Patterns in *global* queries adhere to a virtual graph which refers to classes and properties from arbitrary RDFS or OWL vocabularies. The parser (2) calculates a canonical query plan which is modified by the federator/optimizer component (3). Query federation and optimization is tightly coupled and is described below. The federator analyzes the query and scans the catalog for relevant registered data sources. The output of the optimizer is an optimized global plan which is processed by the

query execution engine (4). Remote sub-plans are executed in a pipelined manner at SPARQL end-points which may be RDF wrappers (5).



## Query Processing



A lexical SPARQL query is parsed into an abstract operator tree using the Jena ARQ library. The generated plan must conform to a slightly modified algebra explained below. The Federator uses ARQ's Transform API to insert *Service* and *Union* operators accordingly. The *Service* operator is an extension of the algebra which executes sub-queries at remote end-points. Query optimization is done within two steps:

1. Transformations based on static rules using JBoss Rules
2. Re-ordering of global join operations based on iterative dynamic programming (IDP)

Because the second step requires the availability of accurate statistics, it can only provide useful results if all end-points involved in a query provide them. Join re-ordering is actually work-in-progress.

A generic RDFStats generator has been partly implemented and will be available as a separate project to allow providers of SPARQL end-points to generate statistics at their local sites which can further be used by SemWIK and other applications. These statistics include instance counts and compressed Base64-encoded histograms for each combination of a class, property, and range used in the data provided at the end-point. The authors will propose a special vocabulary and architecture for end-point statistics soon.

## Query Federation

SemWIK uses a concept-based data integration approach. This means, that all data must be defined as instances of a class. A global query in SemWIK is defined just by  $E'$ , which is a slightly modified SPARQL algebra expression. By contrast to the original SPARQL definition, the dataset  $DS$  is not part of the query (it is virtually defined by the registry catalog) and the only query form  $R$  allowed is *SELECT*.

For  $E'$ , the *Graph* operator was dropped, the special *Service* operator of ARQ2 was added and the basic graph pattern (BGP) is restricted as:  $BGP' = (V \times (I \cup V)) \times (RDF \cup V)$  where  $V$  is the set of query variables,  $I$  the set of IRIs,  $RDF \cup V$  is the set of RDF-Terms. Thus, the subject must always be variable and may never be

concrete. Further, each subject in a query must be typed, i.e. being  $V_s$  the set of all subject variables,  $t$  a triple pattern, and  $I_c$  the set of RDF or OWL classes, the following condition must hold:  $\forall v_s \in V_s \exists t (v_s, rdf:type, c), c \in I_c$

If the vocabulary used in the query (respectively for data descriptions) contains OWL restrictions on properties, the federator may also infer types of linked resources in a query. For the data source selection process, the federator requires at least one asserted or inferred type for each subject variable. Federation leads to a global query plan which is usually very large and requires global plan optimization. The federation algorithm is described in the conference paper.

## Step 1: Rule-based Optimization

As already explained, the first step in plan optimization are several transformations based on statically defined rules. Examples for such rules encoded in JBoss Rules syntax are listed on the right. Because a transformation modifies the query plan, it may trigger further transformations. This process continues until no more rules fire.

One advantage of a rule-based transformation approach is its extensibility. The annotated operator tree (see box below) can be used to decorate ARQ query plans with arbitrary additional information required for further rules.

However, a rule-based approach is not sufficient for global plan optimization. Although simple swap transformations for left/right sub-plans based on cost estimates may be triggered by rules also, it is not performant and may only take local joins into account. An optimal plan may require the complete re-ordering of join operations based on enumeration of all possible plans. This is done in step 2.

```
when
  $left : AnnotatedOp( $lVars : knownVars )
  $right : AnnotatedOp( $rVars : knownVars )
  $join : AnnotatedOpJoin( left == $left && right == $right )
  $filter : AnnotatedOpFilter(
    subOp == $join && $filterVars : filterVars )
eval(!Collections.disjoint($filterVars, $lVars))
eval(Collections.disjoint($filterVars, $rVars))
```

```
then
  move filter beyond join left hand side (similarly, if filter variables are disjoint with variables bound at right operator, move filter down right hand side)
```

```
when
  $service : AnnotatedOpService()
  $op1 : AnnotatedOp1( subOp == $service )
then
  move Op1 beyond OpService
```

```
when
  $filter : AnnotatedOpFilter()
  $subFilter : AnnotatedOpFilter( parent == $filter )
then
  merge filter expressions
```

```
when
  $left : AnnotatedOp( $leftVars : knownVariables )
  $right : AnnotatedOp( $rightVars : knownVariables )
  $unionOp : AnnotatedOpUnion(
    left == $left && right == $right )
  $filterOp : AnnotatedOpFilter(
    subOp == $unionOp && $filterVars : filterVars )
eval(!Collections.disjoint($filterVars, $leftVars))
eval(!Collections.disjoint($filterVars, $rightVars))
```

```
then
  move filter beyond union both sides (similarly, if filter variables are disjoint with left/right variables move down left/right)
```

### Annotated operator tree

The query plan generated by Jena ARQ is a tree of operator nodes. To store additional information for the optimizer as part of the operator tree, the decorator pattern is used. Additional fields and operation may be added to the default ARQ operators. For instance, all decorated operators contain backlinks to their parent operators, known possibly bound variables and operations for the calculation of estimated costs.

## Step 2: Cost-based Join Re-ordering

Basically, all optimization within SemWIK is logical. Physical optimization is usually subject to the local information systems and wrappers. That's why BGPs are not optimized, only operators above *OpService* are relevant since they are executed at the mediator. However, costs of BGP's are still required for overall cost estimates. The most critical factor for the SemWIK query processor is data shipping. The cost-model so far ignores physical data access (possible latencies because of missing indexes at remote end-points) and only relies on the amount of solution bindings produced by an operator. For each operator in the annotated operator tree, a cost estimate can be calculated recursively starting at BGP cost estimates. The basis for these estimates are provided by statistics generated by the RDFStats monitor (see below). Depending on triple pattern shapes

costs are calculated differently:

```
?s a :C .          c = instCount(:C)
now given that ?s refers to instances of :C:
?s :p „val“ .     c = hist(:C, :p, „val“)
?s :p ?o .        c = histAgg(:C, :p)
?s ?p „val“ .     c = Σ hist(:C, p_i, „val“)
?s ?p ?o .        c = Σ histAgg(:C, p_i)
```

These are individual costs for triple patterns. The costs for a BGP is calculated as the minimum of all triple pattern costs. For example:

```
?s a foaf:Person ;
foaf:knows ?who ;
foaf:nick „Jim“ ;
c1 = instCount(foaf:Person) = 124
c2 = histAgg(foaf:Person, foaf:knows) = 420
c3 = hist(foaf:Person, foaf:nick, „Jim“) = 2
```

The cost for the BGP is  $\min(124, 420, 2) = 2$  (the BGP will return 2 solutions). The cost-based join re-ordering is still work in progress. Because it depends on the RDFStats project it will be fully integrated after the release of RDFStats.

## The RDFStats Project

The RDFStats project has been started to provide statistics for remote SPARQL end-points. The data source monitor at the mediator cannot create accurate statistics, it is only used for the data source selection task. The RDFStats monitor creates detailed statistics including instance counts and compressed base64-encoded histograms for all existing combinations of (class, property, datatype/class range).

It is basically part of SemWIK, but it will be released separately because it is regarded as useful for other applications. It could also serve as a basis for future developments regarding end-point descriptions and capabilities. To stay as flexible as possible, the monitor can be run as

a separate process beside a D2R instance or any other SPARQL end-point. It is designed to be placed on the same host or subnet since the generation of histograms requires lots of data to be pulled out of the end-point.

Example of an RDFStats document:

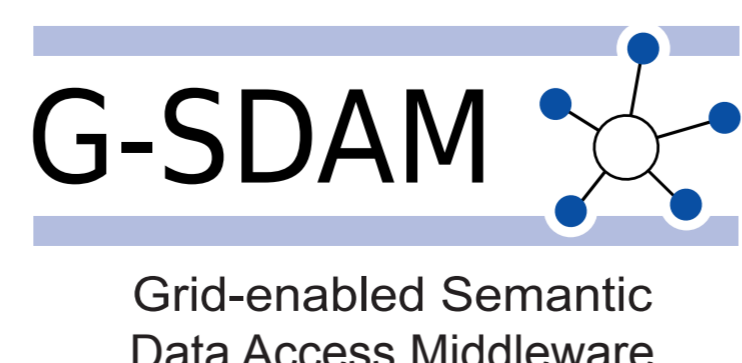
```
ex:c1 stat:iStat 131 ;
  stat:pStat [ stat:forProperty ex:p1 ;
    stat:forRange foaf:Person ;
    stat:hist „AC3A...AAAA=" ] ;
  stat:pStat [ stat:forProperty ex:p1 ;
    stat:forRange xsd:string ;
    stat:hist „C2FA...AA0A=" ] ;
  stat:pStat [ stat:forProperty ex:p2 ;
    stat:forRange xsd:date ;
    stat:hist „3C76...3AAA=" ] .
ex:c2 stat:iCount 20 . (...)
```

For each class several property statistics are created including base64-encoded histograms. The decoding specification will be

Literature references can be found in the corresponding conference paper: A. Langegger, W. Wöß, M. Blöchl: A Semantic Web middleware for Virtual Data Integration on the Web. In Proceedings of the European Semantic Web Conference (ESWC'08), Springer, 2008. This work is part of the Austrian Grid Project, which is funded by the Austrian BMBWK (Federal Ministry for Education, Science and Culture) under contract GZ BMWF-10.220/0002-11/10/2007.



ESWC 2008  
5th European Semantic Web Conference  
June 1-5, Tenerife, Spain



Part of the  
Austrian Grid  
Project



JOHANNES KEPLER  
UNIVERSITÄT LINZ  
Netzwerk für Forschung, Lehre und Praxis