ICT Seventh Framework Programme (ICT FP7)

Grant Agreement No: 318497

Data Intensive Techniques to Boost the Real – Time Performance of Global Agricultural Data Infrastructures

D2.3.4b: Large Scale Distributed Architecture

Deliverable Form	
Project Reference No.	ICT FP7 318497
Deliverable No.	D2.3.4b
Relevant Workpackage:	WP2: Use Cases & Architecture
Nature:	R
Dissemination Level:	PU
Document version:	V5.0
Date:	4/3/2016
Authors:	NCSR-D, SWC, UAH, DLO
Document description:	This report specifies the set of components comprising the *SemaGrow Stack* as well as the components foreseen for its maintenance. It also outlines the technologies involved in each component and foresees the interfaces between them.

Document History

Version	Date	Author (Partner)	Remarks
Draft v0.1	21/1/2013	UAH, NCSR-D	Document Setup
Draft v0.2	25/1/2013	NCSR-D	Initial Contributions
Draft v0.5	28/1/2013	UAH	Internal review
Draft v0.8	31/1/2013	NCSR-D, UAH	Revisions and updates
Draft v1.0	8/2/2013	SWC, UNITOV	Internal review
V 1.0	15/2/2013	NCSR-D, UAH	Delivered as D2.3.1
Draft v1.5	15/7/2013	NCSR-D, SWC	Updates
Draft v1.8	22/7/2013	UAH, UNITOV	Internal review
Draft v1.9	29/7/2013	NCSR-D, SWC	Revisions and updates
V 2.0	31/7/2013	NCSR-D, SWC	Delivered as D2.3.2
Draft 2.1	10/1/2014	NCSR-D, SWC	Intermediate version, prepared for first project review
Draft 2.2	9/10/2014	NCSR-D	Refined architecture with respect to handling heterogeneity (Fig. 1, Sect 2.1; API between Semagrow Stack and alignment, Sect 3)
Draft 2.9	16/10/2104	UAH, UNITOV	Internal review
V 3.0	21/11/2014	NCSR-D	Delivered as D2.3.3
Draft 3.2	26/10/2015	NCSR-D	Refined architecture with respect to resource discovery protocols (Sect 4.2). Updated the overall figure (Fig. 1, Sect 2.1; data source metadata representation and handling of client parameters)
Draft 3.9	27/10/2105	UAH, UNITOV	Internal review
V 4.0	26/11/2015	NCSR-D	Delivered as D2.3.4
Draft 4.1	18/1/2016	NCSR-D, SWC, UAH, UNITOV	Information about the logging mechanism added in new Section 2.2.6. Updated the information flow in case of unresponsive data source (Sect 3.2.2).
Draft 4.2	12/2/2016	NCSR-D	Updated Sect 2.2.5 and Figure 1 (Sect 2.1) to reflect that multiple connections to the same endpoint might be open simultaneously
Draft 4.3	12/2/2016	NCSR-D, DLO	Updated Sections 2.2.2 to document changes in the abstract query representation necessitated by the DLO use case
		UAH, UNITOV	Internal review
V 5.0		NCSR-D	Delivered as D2.3.4b

EXECUTIVE SUMMARY

This report specifies the set of components comprising the SemaGrow Stack as well as the components foreseen for its population, maintenance, and demonstration. It also outlines the technologies involved in each component and foresees the interfaces between them. The document was developed as a *living document*, in which the system's architecture and interfaces were documented as they evolved from new insights gained during the project. This final version (Deliverable 2.3.4) provides an overview of the components of the system and of the information flow between them.

TABLE OF CONTENTS

LIST OF TERMS AND ABBREVIATIONS

Term/Abbreviation	Definition
Data source	In the context of this document, one of the *SPARQL endpoints* federated by the *SemaGrow Stack*. Besides their location, the *SemaGrow Stack* also retains metadata about its data sources' contents and schema.
Query pattern	SPARQL queries are made up of *query patterns*, subject-predicate-object triples of either specific values or variables. Triple patterns are sometimes also called *triple patterns*.
Querying	Querying retrieves a solution sequence that corresponds to the ways in which the query's pattern matches the data that is being queried.
RESTful	A *RESTful Web Service* (also called a *RESTful Web API*) is a web service implemented using HTTP and the principles of *REST*, the architectural style of stateless Web Services.
RDF	The *Resource Description Framework (RDF)* is a W3C Recommendation for a language for representing information about resources in the World Wide Web. RDF statements are *triples*; data entities composed of subject, predicate, and object.
SPARQL	*SPARQL* is a W3C Recommendation for expressing queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. The results of SPARQL queries can be result sets or RDF graphs.
SPARQL endpoint	A *SPARQL endpoint* is a machine-friendly (but also human-usable) service that enables clients to query an RDF knowledge base via the SPARQL language.
SPARQL endpoint federation	A *SPARQL endpoint federation* provides a single endpoint through which to query distributed remote SPARQL endpoints. Federations are queried in the *SPARQL Federated Query Extension*, a W3C Proposed Recommendation.
Triple store	A *triple store* is a database that is optimized for the storage and retrieval of RDF triples.
VoID	The *Vocabulary of Interlinked Datasets* is an RDF Vocabulary for expressing metadata about Datasets published as Linked Open Data.

1. INTRODUCTION

1.1 Purpose and Scope

This report concerns the technical requirements for the operation of the components of SemaGrow, a generic large scale and distributed architecture.
At the core of the SemaGrow architecture is the concept of efficiently identifying the optimal way to distribute a query among the nodes of a federation of SPARQL endpoints over heterogeneous data sources. The SemaGrow architecture proposes that this is achieved by means of collecting and indexing meta-information about the data stored in each data source; in this manner the data sources do not need to be cloned and re-hashed, and the way data is distributed among them does not need to be centrally controlled.
This report documents and defines the set of services, standards, and technologies to be brought together in order to realize this concept into a scalable Service Oriented Architecture specification for (a) the collection and indexing of data summaries and other data source annotations, and (b) the usage of such source annotations for source selection and query decomposition in distributed querying.

1.2 Approach to Work Package

This document was developed as a *living document*, in which the system's architecture and interfaces was documented as they evolved from new insights gained during the project.
In the remainder of this document, we first provide an overview of the *SemaGrow Stack* (Section 2). We then proceed to specify the components of the stack and the information flow between them (Section 3) and the interfaces between the stack and external tools (Section 4).
Although the core architectural decisions have not changed from the previous deliverable versions, some revision on passing logging and contextual information have been made (c.f. Section 2.2.6).

1.3 Relation to other Work Packages and Deliverables

The system architecture was developed in Task 2.3 is based on work on Task 2.1 Envisaged Applications and Use Cases and guides work in the core technical work packages (WP3 and WP4) as well as integration work (WP5).
The relation with work packages WP3, WP4, and WP5 is bi-directional, as architecture refinements and optimisations are driven by the experience gained while implementing the previous version of the architecture. This bidirectional relation is also evidenced by the fact that Task 2.3 delivers the final version of the architecture after all components have been developed, integrated and deployed (M33).
In relation to the previous version of the architecture (D2.3.3) the following changes have been effected:

- Section 4.2 and the figure of the architecture have been updated to reflect that metadata is now represented in the Sevod vocabulary developed within WP3 during Y3.
- Section 4.2 has been updated to document the handling of client parameters
- Section 2.2.2 has been updated to explain the information added by *resource discovery* and *query decomposition* to the Abstract Syntax Tree that is produced by the Sesame parser
- Section 2.2.2 has been updated to reflect that in order to correctly handle federating a spatio-temporal store in the DLO pilot (and in general extensions to standard SPARQL) user-defined functions must be pushed to the leaves of the AST
- Section 2.2.5 and the figure of the architecture have been updated to reflect that with the new reactive execution engine, multiple instances of the executor might connect to the same remote endpoint
- New Section 2.2.6 has been added to document the new logging mechanism developed within WP4 and WP5 during Y3.
- Section 3.2.2 has been updated to document behaviour in case of unresponsive data sources (Section 3.2.2)

2. SemaGrow Architecture

2.1 Overview

This section provides an overview of the components that comprise the *SemaGrow Stack* (Figure 1) and the associated maintenance tools.

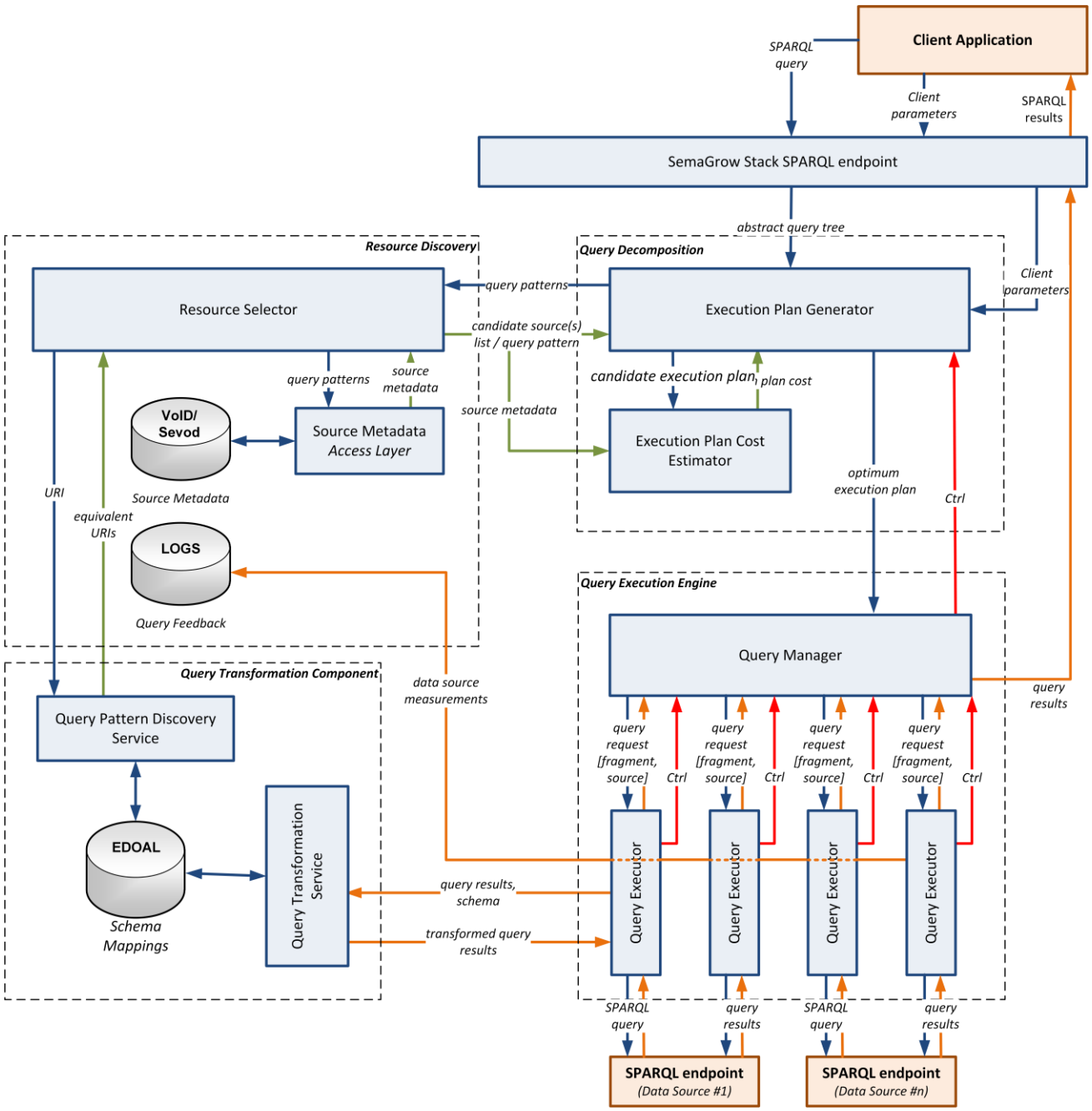

Figure 1: The SemaGrow Stack

2.2 The SemaGrow Stack

The *SemaGrow Stack* integrates the components needed in order to offer a single SPARQL endpoint that federates a number of heterogeneous data sources, also exposed as SPARQL endpoints. The main difference between the *SemaGrow Stack* and most existing distributed querying solutions is that *SemaGrow* targets the federation of heterogeneous and independently provided data sources. In other words, *SemaGrow* aims to offer the most efficient distributed querying solution that can be achieved without controlling the way data is distributed between sources and, in general, without having the responsibility to centrally manage the data sources of the federation.

2.2.1 SemaGrow SPARQL endpoint

The *SemaGrow SPARQL endpoint* is the Web service through which client applications access SPARQL endpoints that have been federated using the *SemaGrow Stack*.

2.2.2 Query Decomposition Component

The *query decomposition* component analyses SPARQL queries and identifies the various ways in which they can be broken up into query fragments to be dispatched to sources' endpoints. The *query decomposition* component comprises the *query decomposer plan generator* and the *execution plan cost estimator*.

The *query decomposer* syntactically analyses queries and requests (from the *resource discovery* component) for each pattern in the query information regarding the data sources that can provide matching triples; including data sources where a vocabulary translation is needed before they can be queried. It then prepares a list of ways to partition the query into fragments of one or more query patterns along with the data sources that can be contacted for each fragment. The *query execution plan generator* constructs valid execution plans and then assigns for each plan a cost that is received from the *execution plan cost estimator*. The plan generator evaluates these suggestions to select the ones that appear optimal, combining *static strategies* (such as preferring longer fragments in order to minimize joining results from different data sources) and *dynamic client parameters*. *Client parameters* include:

- *reactivity parameters* that specify the client application's wished position in the trade-off between efficiency and completeness in terms of how much time it is possible and worth it to wait to get more results, or the minimum number of results required, or some other similar policy balancing between completeness and effort.
- *trust parameters* that specify the client's trust or any other kind of prior preference towards the data sources of the federation.

The execution plan with the least assigned cost is sent to the *query execution engine* for execution.

The changes with respect to the abstract query that is output of the SPARQL string parser are the following:

- The abstract syntax tree has been re-arranged so that the ordering of child nodes reflects the order of execution
- Some nodes have been annotated with endpoint URLs, designating that the sub-tree under that node should be executed at a remote endpoint
- When a vocabulary transformation is required, this is also recorded
- Abstract operators (e.g. "Join") have been replaced by specific implementations of the operator (e.g., "BindJoin" or "MergeJoin").
- FILTERS and user-defined functions have been pushed to the leaves. This is particularly important for user-defined functions that can only be executed at specific members of the federation, such as spatial extensions that can only be executed at GIS systems.

This representation includes all information required in order to execute the query.

2.2.3 Resource Discovery Component

The *resource discovery* component receives a query pattern and responds with a list of data sources that are likely to hold matching triples; including sources that follow a different (but aligned) schema than that of the query pattern. The resource discovery component comprises a *resource selector* module and a *source metadata access* module.

The *source metadata access* module manages access to the source metadata repositories and is able to respond to individual URI resources with the estimated number of triples that mention this URI resource in each of the data sources of the federation. Although in principle any repository infrastructure can be used to store these annotations, it is one of the core research objectives of *SemaGrow* to experiment with the *Vocabulary of Interlinked Datasets (VoID)* in order to take advantage of naming convention regularities to compress such indexes.

The *resource selector* module uses this information in order to estimate the number of triples that match a given query pattern. Furthermore, *resource selector* combines ontology alignment information it receives from the *query transformation* component in order to also include sources that hold matching triples following a different (but aligned) schema.

The result is that *resource discovery* responds to *query decomposition* for each query pattern with a list of data sources annotated with: (a) the estimated number of triples in the data source that match the given query pattern (including data sources where a vocabulary translation is needed before they can be queried); (b) the *semantic proximity* of the transformation, a numerical valuation of how closely a transformation preserves the semantics of the original terms, provided by *query transformation*; and (c) a numerical valuation of the source's average responsiveness.

2.2.4 Query Transformation Component

The results of the Ontology Alignment module are stored in the Schema Mappings repository and subsequently used by the *Query Transformation* component, which provides two services: *query pattern discovery* and *vocabulary translation*, both serving vocabulary mapping knowledge generated by the *ontology alignment tool*.

The *query pattern discovery service* responds to URI resources with a list of possible translations into other schemas. This is used by *resource discovery* in order to extend the list of potential sources with those that use a different (but aligned) vocabulary than the one of the original query. For each possible translation, it also provides an identifier for the *transformation model* and the model's *semantic proximity*, a numerical valuation of how closely a transformation preserves the semantics of the original.

The *vocabulary translation* service applies a *transformation model*, expressed in the *Expressive and Declarative Ontology Alignment Language (EDOAL)*, to re-write query fragments from the schema of the original query to that of the data source that will be used for each fragment and also query results back into the schema of the original query so that they can be joined with results from other sources. To this end, the service defines the necessary functions to obtain a (near) equivalent resource under a given transformation model, for each of the resources contained in the input fragment.

2.2.5 Query Execution Engine

The Query Execution Engine manages the communication with the external data sources that are federated by the *SemaGrow Stack*. It comprises two modules, the *query manager* and the *query executor.*

The *query manager* receives from the *query decomposition* component an *execution plan*, the detailed instructions on how the query should be executed specified in Section 2.2.2 above. Based on this, as well as dynamic information about the federated data sources (such as response time and current availability), the query manager uses and orchestrates multiple instances of the *query executor* module to execute the plan it received. Executor instances are relatively short lived, so (for efficiency) they are re-used from a pool of available executor threads rather than instantiating new ones. Query executors are responsible for using, where necessary, the *query transformation service* to access repositories that follow a different schema than the one of the original query.

Since the executors directly accesses the federated endpoints, they are also responsible for updating the data source availability information, observing that querying parameters are satisfied to the extent possible, and using the control channel to request from the *query manager* the formulation of a new execution plan if the current is impossible to realize

due to source unavailability. The executors also collect *data source measurements* by observing the results stream, which are forwarded to *resource discovery*. These measurements are used by *resource discovery* in order to update the *instance and responsiveness statistics* that it maintains about the federated endpoints.

The *query manager* orchestrates the merging of results from each executor into the overall result. The *query manager* also uses the control channel to request a new *query decomposition plan* from *query decomposition*, if the current one cannot be executed due to source unresponsiveness.

2.2.6 Logging

Orthogonal to the interactions of the several components of the SemaGrow stack architecture, all components emit logging information about the actions and decisions made during a complete query processing. A standard logging interface is shared through the several components of the stack and can be used to store logging entries. Moreover, these logging entries are associated with contextual information about the user, the query, and the environmental parameters during execution. It should be clarified that "query" in the above does not pertain to distinct query strings, but to each specific query request, and the log lines are annotated with a query ID generated when the client first posts the query to the SemaGrow endpoint. What should be noted is that the query context (including this ID) must be passed from the query manager to the executors when the manager assigns an executor from the pool to a query.

The logs are analysed and visualized by the rigorous testing components to extract performance data and by the system health monitoring components to visualized current performance.

2.2.7 Overview

To recapitulate and clarify the interaction between these components, handling heterogeneity and distribution is broken down in the following three layers:

- At the level of individual URI resources, *Query Pattern Discovery* handles schema heterogeneity by providing mappings to (near) equivalent URI resources and *Source Metadata Access* handles distribution by providing references to data sources containing triples where URI resources are mentioned, regardless of their position in the triple.
- At the level of individual triple patterns, *Resource Selector* handles both heterogeneity and distribution by combining the information received from *Query Pattern Discovery* and *Source Metadata Access* into a list of sources where triples can be found that match the triple pattern (or a schema-mapped semantic equivalent).
- At the level of a complete query, the *Query Decomposition* and *Query Execution Engine* components handle both heterogeneity and distribution by combining the information received from *Resource Selector*, the reactivity and completeness requirements set by the application posing the query, and the run-time responsiveness of the federated end-points at the query time.

2.3 Maintenance components

The system also foresees update and maintenance cycles, where new end-points are added to the federation or update the schema they employ or have accumulated considerable changes in the instances they hold. Schema metadata must be provided by the data provider when joining the federation, using authoring tools and tutorials produced by the project. Instance metadata may also be provided, but are also automatically maintained by the *resource discovery and query decomposition component* based on statistics extracted from query results.

2.3.1 Authoring Tool

The *semantic annotation environment* is the visual tool that will assist data providers with implementing the guidelines for adding their data source to a SemaGrow federation. This includes cataloguing metadata such as the location of SPARQL endpoint and the schemas used, as well as more detailed instance-level metadata about the specific resources described by or mentioned in the dataset.

2.3.2 Ontology Alignment Tool

The *ontology alignment tool* will be responsible for aligning the various semantic vocabularies used by data providers and consumers. The component will present alignment suggestions to the data source providers to accept, modify, or reject. Accepted alignments will be recorded in the *schema mappings repository* from where to be queried by the *SemaGrow Stack* components. The *ontology alignment tool* will assist data source providers to introduce new data sources, but also to integrate updates in the schema used in an existing source.

2.3.3 Content Classification and Ontology Evolution

Content classification and *ontology evolution tools* will also be used when new data sources are added. They will be used to refine coarsely annotated data and to bring annotations to a level of detail where they can be more accurately aligned with other schemas used in the federation.

3. INTERNAL INTERFACES

In this section, we first describe the interfaces between the components of the *SemaGrow Stack* and then present the various information flows and the external or internal events that trigger each flow

3.1 Component Interfaces

The SemaGrow SPARQL endpoint:

- Receives from client applications SPARQL queries and client parameters (reactivity and trust parameters).
- Receives from the Query Execution Engine SPARQL query results.
- Sends to client applications SPARQL query results.
- Sends to Query Decomposition SPARQL queries.
- Sends to Query Decomposition client parameters.

Query Decomposition:

- Receives from the SemaGrow SPARQL endpoint SPARQL queries and client parameters.
- Requests from Resource Discovery source metadata with respect to a given query pattern. Resource Discovery responds with instance statistics (the approximate number of triples in each data source that match the pattern, including triples that match under a vocabulary translation), the semantic proximity of the translation, and a measure of the average or usual responsiveness of the data source.
- Sends to the Query Execution Engine a decomposition plan in a formalism that extends the Sesame *Abstract Syntax Tree (AST)* so that it also encodes: a partition of the query into fragments and the source that each fragment is to be dispatched to; and the vocabulary translation that needs to be applied to each fragment, if any.
- Receives from the Query Execution Engine a control signal regarding the inability to implement an execution plan, to which it responds with a new plan.

The Query Execution Engine:

- Receives from Query Decomposition an AST that encodes the execution plan and relevant parameters.
- Sends to Query Decomposition a control signal regarding the inability to implement an execution plan.
- Requests from Query Transformation to apply a transformation model to either a query or to query results.
- Queries the endpoints of the federation.
- Sends to the SemaGrow SPARQL endpoint the SPARQL query results.

Resource Discovery:

- Receives data source measurements from the Query Execution Engine.
- Requests from Query Transformation known mappings of a given URI resource. Query Transformation responds with a list of (near) equivalent URI resources, an identifier for the relevant transformation, and the semantic proximity of this transformation.
- Responds to Query Decomposition request for source metadata with respect to a given query pattern.

Query Transformation:

- Responds to Resource Discovery requests for mappings of a given URI resource.
- Applies transformation models to resources included in the records within a query result set.

3.2 Information Flow

In this section we describe the information flow triggered by a client application's executing a query, and exemplify it using a scenario.

3.2.1 Example scenario

Usage scenario that illustrates how the SemaGrow Stack components and other SemaGrow tools are used.

Scenario	Semagrow Stack	Other SemaGrow Tools
A client application provides educational resources and "further reading" relevant scientific publications by joining results between: • The AGRIS bibliography dataset • The Organic.Edunet educational resources In order to configure a SemaGrow stack to federate these datasets we need their descriptions: • The VoID description of AGRIS is downloaded from the AGRIS site • ELEON is used to manually author a very rough description of Organic.Edunet	Federated querying of two endpoints that have different properties of the same instances (joining across datasets) Configuration via existing and minimal, manual dataset description	ELEON: Authoring simple SemaGrow configuration
In order to improve performance, the *metadatagen* tool can create a more detailed description of Organic.Edunet	Detailed dataset description allow optimizer to producer better execution plans	Metadatagen: Automatically creating detailed SemaGrow configuration
Organic.Edunet uses a different schema than AGRIS, so we must also configure SemaGrow with schema mappings The mappings are semi-automatically produced by SYNTHSIS and Semantic Turkey and loaded into the SemaGrow Stack.	Application of schema mappings	SYNTHESIS: Automatically generated mappings between the schemas used by AGRIS and Organic.Edunet Semantic Turkey's Alignments Validation Workbench: Manual validation and refinement of the SYNTHESIS output.

3.2.2 Resource discovery and query decomposition

Queries that are received by the *SemaGrow SPARQL endpoint* are forwarded to *Query Decomposition*, which first extracts all triple patterns from the query.

Query Decomposition produces and evaluates alternative ways to decompose queries into pairs consisting of query fragments and the data source where each fragment will be executed. During this, *Query Decomposition* sends query patterns to *Resource Discovery*, which responds with candidate data sources for each pattern and annotations pertaining to the source schema, volume of matching triples, and current source load. *Resource Discovery* queries the *data summaries* and the *schema mapping repositories* for this information; *Resource Discovery* is informed of the data source's current load via the control channel from the *Query Execution Engine*.

Upon deciding about the optimal way to break up the query into fragments, these are forwarded to the *Query Execution Engine*. Where necessary, the *Query Execution Engine* uses information from the *schema mapping repository* to translate the fragment into one using the same vocabulary as the source. Then, the *Query Execution Engine* initializes query connections with all relevant sources and starts collecting responses; the *Query Execution Engine* might notify *Query Decomposition* of problems such as unavailable sources and request an alternative query decomposition that does not rely on such sources. In the case of an unresponsive data source the Query Execution Engine terminates the query evaluation and respond to the coordination control level with the specific data source that became unresponsive. Meanwhile, the Resource Discovery component is also informed that triggers the exclusion of the specific source for

subsequent query decompositions. The Query Decomposition procedure is then restarted in order to produce a new execution plan for the current query that will not contain the unresponsive data source.

Once queries have started executing on the endpoints, the *Query Execution Engine* receives the result triples as they become available and (where necessary) translates them back into the vocabulary of the original query. In this incremental receipt, non-blocking join operators applied on triples merge individual parts of the queries into response tuples. The query decomposition execution plan, together with the join operators themselves, allow the incremental handling of massive result sets, allowing the use of secondary memory. Temporarily blocked sources and delays are deferred by returning tuples as they become available without hurting the final result set. This approach is also applicable for streams and, aggregate queries, and in general all contexts foreseen by the SemaGrow use cases.

The results are finally sent to the *SemaGrow SPARQL endpoint* to be forwarded to the client application.

It should be noted that during the execution of the query, diagnostic information is also shared by the components for logging purposes. Each component of the stack emits logging entries about the tasks performed in each stage of the query. Those entries are collected, as usual, to a logging storage, and are processed offline for monitoring the performance of the stack.

4. EXTERNAL INTERFACES

In this section we specify the interfaces between the stack and components outside the stack, both maintenance components developed within the SemaGrow project and external applications and repositories.

4.1 Client Applications

The *SemaGrow Stack* exposes a SPARQL 1.1 endpoint, in particular supporting the recommendation's federated querying capabilities.[1] Pushing beyond the recommendation and the state of the art, the *SemaGrow Stack*'s endpoint is able to automatically distribute among the nodes federated under the *SemaGrow Stack* without requiring that the user tells the system where to dispatch each query fragment using the SERVICE keyword.

Furthermore, the *SemaGrow Stack* supports different retrieval models, such as requiring complete results or preferring to get results as quickly as possible even if incomplete. Besides the LIMIT keyword, it might also be decided to use syntax extensions (keywords) or magic predicates in order to provide more detailed client parameters, including reactivity parameters such as 'retrieve all the triples that can be fetched within a given time limit, as long as they are at least so many' and trust parameters that exclude given endpoints from being used.

The *SemaGrow Stack* does *not* support INSERT, DELETE and, in general, does not provide any means of updating the data held in the federated sources. The update and maintenance of the federated sources is the responsibility of the source manager and is outside the scope of the *SemaGrow Stack* and the project.

4.2 Source Descriptions

The *SemaGrow Stack* needs access to metadata describing each source and its contents. These stores are internal to the stack (as described in the previous section), but the information in them is provided and maintained by the source managers. Here we describe the formats and interfaces used to edit this metadata.

4.2.1 Source-level metadata

To be able to describe members of the federation in a machine readable way SemaGrow make use of the Vocabulary of Interlinked Datasets (VoID). Description of members of the federation include information on the RDF schemes or ontologies in use by the member resulting in means to retrieve a complete definition of a member of the federation unambiguously.

```
:DBpedia a void:Dataset;
    void:uriRegexPattern "^http://dbpedia\\.org/resource/";
    void:vocabulary <http://dbpedia.org/> ;
    sd:Language sd:SPARQL11Query.
```

4.2.2 Instance-level summaries

Instance-level summaries support *Resource Discovery* by providing the information needed in order to decide which source a query pattern can best be dispatched to for execution. Conceptually, an instance-level summary holds the complete *P* x *S* x *M* relation, where:

4.2.3 New source added to the federation or source metadata updated

In order to take full advantage of the advanced *heterogeneous querying* and *query decomposition* facilities in the SemaGrow Stack, data sources need to declare the schema that formalizes the data they serve, as well as instance-level metadata about the contents of the repository. Although this can be done by directly authoring such metadata, *SemaGrow* will also provide a specialized *visual authoring tool* that will assist data providers to author and maintain such

[1] *SPARQL 1.1 Federated Query*, W3C Recommendation, 21 March 2013. URL http://www.w3.org/TR/sparql11-federated-query

metadata. This process updates the data summaries repository and the updated metadata are immediately available to all components served by the *data summaries endpoint*.

4.2.4 Bulk import and export

Data summaries for a given source can be bulk imported into and exported from a *SemaGrow Stack* deployment using an extension of the VoID vocabulary. The Vocabulary of Interlinked Datasets (VoID) is a Semantic Web Vocabulary for expressing metadata, statistics and links between datasets. VoID allows us to express the knowledge that certain endpoints serve datasets satisfying certain constraints. For example, one can define the DBpedia dataset that contains only entities with URIs of a particular form. Moreover, someone can define the interlinking between datasets via appropriate predicates and in addition define statistics such as total size of the dataset.

The original VoID Recommendation defines a series of `<Dataset/>` properties facilitating the expression of groupings of resources (thus partitions of the original Dataset, which are themselves Datasets) in the resolvable Web. On the other hand, VoID does not provide any more expressive filtering expressions on subjects or object, or specify selectivity properties between a pair of Datasets. In order to express the data summaries needed by the *Semagrow Stack*, we have defined several new properties attached to a VoID Dataset. The complete and formal definition is in Deliverable 3.1.

It should be noted that this is *not* the data model used in order to internally store data summaries, but rather its human-readable serialization for importing and exporting.

This serialization can be edited either in general-purpose XML editors or using the specialized *Synergetic Semantic Annotation Environment* developed in Task 5.2, supporting data providers with all the configuration and metadata authoring needed in order to include their repositories into the *SemaGrow* federation.

4.3 Vocabulary Mappings

The SemaGrow Stack - and specifically the Query Pattern Discovery service and the Query Transformation service – use the mappings between the schemas of the federated collections repositories in order to perform their respective procedures. The alignment process is performed offline, that is, the flow of the SemaGrow Stack does not incorporate any activation of the alignment component, but rather uses the alignment results that are stored in the respective repository (see Figure 1).

4.3.1 Vocabulary mappings

Like the majority of the state-of-the-art alignment systems, the mappings produced will be 1:1 correspondences between the elements of the compared schemas. They refer to both classes and properties of the ontologies that are compared. Each mapping bears a confidence score, in the (0, 1] range. The mappings that should be taken into account from the relevant SemaGrow components are the ones declared explicitly within the repository. Transitivity and, more generally, other algebraic operations over the produced alignments, is a subject of examination. In general, the calculation of the confidence for mappings deducted is not intuitive and depends on the process employed by each alignment method. In any case, the system will be designed in a way that will allow the future support of some algebra over the produced ontology alignments.

4.3.2 New alignment

Besides declaring which schema a source follows, it is also needed that schemas are (where semantically feasible) aligned, in order to facilitate the querying of heterogeneous sources. Alignment is a semi-automatic process, where computational intelligence methods are used to discovery and suggest alignment knowledge to a human validator.

The resulting knowledge is stored in the *schema mappings repository* and the updated metadata will be immediately available to all components through that repository's endpoint.

4.3.3 Bulk upload and download

The representation of ontology alignment information will follow the Ontology Alignment format[2], generally regarded as the standard formalization used by the relevant community. For purposes of further extensibility, the format will follow the EDOAL paradigm[3].

The root element for a particular alignment is the Alignment element. For each alignment, the compared ontologies, the arity of the alignment and the mappings defined by the alignment are described. Each alignment contains a set of correspondences (Cells). Each Cell describes the entities that are mapped, their relation (subsumes, is subsumed, is equivalent) and the confidence for the particular association. A full-fledged example of a hypothetical alignment follows. The alignment entails two mappings between elements of the two ontologies; one subsumption and one equivalence relation between classes. For readability purposes, namespaces have been omitted from the example.

4.4 Sources

The *SemaGrow Stack* federates data sources that expose SPARQL end-points. The *SemaGrow Stack* does not delve into any further details regarding the implementation and the update and maintenance procedures of these sources.

[2] http://alignapi.gforge.inria.fr/format.html

[3] http://alignapi.gforge.inria.fr/edoal.html

```
<Alignment>
  <xml>yes</xml>
  <level>0</level>
  <type>11</type>
  <method>eu.semagrow.alignment.method</method>
  <time>10</time>
  <onto1>
   <Ontology rdf:about="http://www.example.org/ontology1">
      <location>file:examples/rdf/onto1.owl</location>
      <formalism>
        <Formalism                                    align:name="OWL1.0"
align:uri="http://www.w3.org/2002/07/owl#"/>
      </formalism>
    </Ontology>
  </onto1>
  <onto2>
   <Ontology rdf:about="http://www.example.org/ontology2">
      <location>file:examples/rdf/onto2.owl</location>
      <formalism>
        <Formalism                                    align:name="OWL1.0"
align:uri="http://www.w3.org/2002/07/owl#"/>
      </formalism>
    </Ontology>
  </onto2>
  <map>
    <Cell>
      <entity1
rdf:resource='http://www.example.org/ontology1#reviewedarticle'/>
      <entity2
rdf:resource='http://www.example.org/ontology2#journalarticle'/>
      <relation>></relation>
      <measure
rdf:datatype='http://www.w3.org/2001/XMLSchema#float'>0.8</align:measure
>
    </Cell>
  </map>
  <map>
    <Cell>
      <entity1
rdf:resource='http://www.example.org/ontology1#journalarticle'/>
      <entity2
rdf:resource='http://www.example.org/ontology2#journalarticle'/>
      <relation>=</relation>
      <measure
rdf:datatype='http://www.w3.org/2001/XMLSchema#float'>1.0</measure>
    </Cell>
  </map>
</Alignment>
```

Figure 2: An exemplary alignment described in the Ontology Alignment Format