

QAESTRO – Semantic-Based Composition of Question Answering Pipelines

Kuldeep Singh^{1,2}, Ioanna Lytra^{1,2(✉)}, Maria-Esther Vidal¹,
Dharmen Punjani³, Harsh Thakkar², Christoph Lange^{1,2}, and Sören Auer^{1,2}

¹ Fraunhofer Institute for Intelligent Analysis and Information Systems (IAIS),
Sankt Augustin, Germany

`Kuldeep.Singh@iais.fraunhofer.de`, `{lytra,langec,auer}@cs.uni-bonn.de`

² Institute for Applied Computer Science, University of Bonn, Bonn, Germany
`{vidal,thakkar}@cs.uni-bonn.de`

³ Department of Informatics and Telecommunications,
National and Kapodistrian University of Athens, Athens, Greece
`Dharmen.punjani@gmail.com`

Abstract. The demand for interfaces that allow users to interact with computers in an intuitive, effective, and efficient way is increasing. Question Answering (QA) systems address this need by answering questions posed by humans using knowledge bases. In recent years, many QA systems and related components have been developed both by practitioners and the research community. Since QA involves a vast number of (partially overlapping) subtasks, existing QA components can be combined in various ways to build tailored QA systems that perform better in terms of scalability and accuracy in specific domains and use cases. However, to the best of our knowledge, no systematic way exists to formally describe and automatically compose such components. Thus, in this work, we introduce QAESTRO, a framework for semantically describing both QA components and developer requirements for QA component composition. QAESTRO relies on a controlled vocabulary and the Local-as-View (LAV) approach to model QA tasks and components, respectively. Furthermore, the problem of QA component composition is mapped to the problem of LAV query rewriting, and state-of-the-art SAT solvers are utilized to efficiently enumerate the solutions. We have formalized 51 existing QA components implemented in 20 QA systems using QAESTRO. Our empirical results suggest that QAESTRO enumerates the combinations of QA components that effectively implement QA developer requirements.

1 Introduction

The main goal of Question Answering (QA) systems is to allow users to ask questions in natural language, to find the corresponding answers in knowledge bases, and to present the answers in an appropriate form. In recent years, QA systems have received much interest, since they manage to provide intuitive interfaces to humans for accessing distributed knowledge – structured, semi-structured, or unstructured – in an efficient and effective way. Since the first

attempts to provide natural language interfaces to databases around 1970 [1], an increasing number of QA systems and QA related components have been developed by both industry and the research community [12, 16].

Despite different architectural components and techniques used by the various QA systems, these systems have several high-level functions and tasks in common [22]. For instance, the analysis of a question often includes tasks such as named entity recognition, disambiguation, and relation extraction, to name a few. Recent literature reviews have studied and classified existing QA systems and QA components with regard to the tasks they attempt to solve [22] and the common goals and challenges they tackle [12, 16]. In addition, several frameworks have been proposed to address the re-usability of QA components. For instance, openQA [17] suggests a modular QA system consisting of components performing QA tasks that expose well-defined interfaces. The interchangeability of QA components is the main focus of other approaches as well, such as QALL-ME [8] which proposes a service-oriented architecture for the composition of QA components and Qanary [19] which introduces an ontology to tackle interoperability in the information exchange between QA components.

The aforementioned works provide a framework for developing or even integrating QA systems but fail to systematically address how to formally describe and automatically compose existing QA components. Still the composition of new QA systems for a specific domain or use case given the plethora of existing QA components is a rather manual, tedious, and error-prone task.

We introduce QAESTRO, a framework to semantically describe QA components and QA developer requirements and to produce QA component compositions based on these semantic descriptions. In particular, we introduce a controlled vocabulary to model QA tasks and exploit the *Local-As-View* (LAV) approach [15] to express QA components. Furthermore, QA developer requests are represented as conjunctive queries involving the concepts included in the vocabulary. The QA Component Composition problem can be afterwards cast to the LAV *Query Rewriting Problem* (QRP) [11]. Then, state-of-the art SAT solvers [10] can find the solution models in the combinatorial space of all solutions which eventually correspond to QA component compositions. Using QAESTRO, we formalized 51 QA components included in 20 distinct QA systems. In an empirical study, we show that QAESTRO effectively enumerates possible combinations of QA components for different developer requirements. Our main contributions are: (1) a vocabulary for expressing QA tasks and developer requirements; (2) the formalization of existing 51 QA components from 20 QA systems; (3) a mapping of the QA Component Composition problem into QRP; (4) the QAESTRO framework that generates QA component compositions based on developer requirements and (5) an empirical evaluation of QAESTRO behavior on QA developer requirements over the formalized QA components.

The remainder of the paper is structured as follows. We introduce the problem of QA Component Composition in the context of a motivating example in Sect. 2. In Sects. 3 and 4, we introduce the QAESTRO framework and present its details

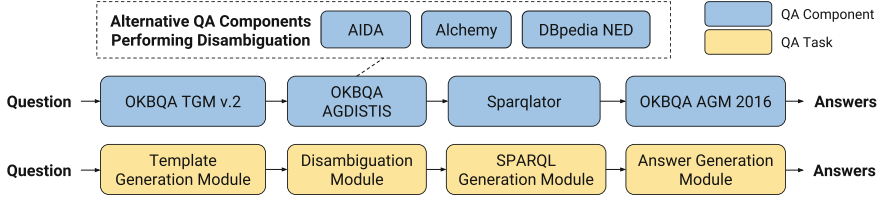


Fig. 1. OKBQA QA Pipeline and Pipeline Instance. OKBQA pipeline consists of four components that implement four core modules: Template Generation Module, Disambiguation Module, Query Generation Module, and Answer Generation Module. In this example, the disambiguation task can be performed by OKBQA AGDISTIS, Alchemy, and DBpedia NED interchangeably.

respectively. The results of our evaluation are reported in Sect. 5. We discuss the related work in Sect. 6 and conclude with an outlook on future work in Sect. 7.

2 Motivating Example

We motivate our work by discussing the problem of QA component composition in the context of the Open Knowledge Base and Question Answering (OKBQA) framework¹. OKBQA considers QA as a predefined workflow consisting of four core modules providing Web service interfaces: (1) Template Generation Module for analyzing a question in natural language and producing SPARQL query skeletons, (2) Disambiguation Module for mapping words or word sequences to Linked Data resources, (3) Query Generation Module for producing SPARQL queries based on modules (1) and (2), and finally, (4) Answer Generation Module for executing SPARQL queries to get the answers. Figure 1 illustrates an instantiation of a QA pipeline with the components OKBQA TGM v.2, OKBQA AGDISTIS, Sparqlator, and OKBQA AGM 2016 which implement the aforementioned modules (1)–(4), respectively². Although OKBQA provides a public repository comprising several QA components that can be composed in the OKBQA pipeline, still several issues remain open for the QA system developer. First of all, there is no systematic way to identify other existing components – either standalone or parts of other QA systems – that could be part of the OKBQA pipeline. Secondly, there is no way to exploit OKBQA QA components in existing QA systems systematically. Thirdly, it is not clear whether and how other QA-related tasks and/or subtasks can be integrated in the OKBQA framework. For instance, let us consider the disambiguation task. Several components, such as Alchemy API³, and DBpedia NED [18] may replace OKBQA AGDISTIS in the QA pipeline of Fig. 1 since they perform conceptually the same QA task. Similarly, OKBQA AGDISTIS could serve the purpose of disambiguation

¹ <http://www.okbqa.org/>.

² All components can be found at <http://repository.okbqa.org>.

³ <http://alchemyapi.com>.

in other QA systems as well. The same observation holds for other QA tasks that can participate in a QA pipeline. Hence, with the growing number of QA components, identifying all viable combinations of QA components that perform one or more tasks in combination requires a complex search in the large combinatorial space of solutions, which until now has to be performed manually.

3 QAESTRO Framework

QAESTRO is a QA framework that allows for the composition of QA components into QA pipelines. QAESTRO is based on a QAV vocabulary, which encodes the properties of generic QA tasks and is utilized to semantically describe QA components. QAESTRO exploits semantic descriptions of QA components, and enumerates the compositions of the QA components that implement a given QA developer requirement. Thus, QAESTRO provides a semantic framework for QA systems that not only enables a precise description of the properties of generic QA tasks and QA components, but also facilitates composition, integration, and reusability of semantically described QA components.

Formally, QAESTRO is defined as a triple $\langle \text{QAV}, \text{QAC}, \text{QACM} \rangle$, where: *(i)* QAV is a domain vocabulary composed of predicates describing QA tasks, e.g., disambiguation or entity recognition; *(ii)* QAC is a set of existing QA components that implement QA tasks, e.g., AGDISTIS [23] or Stanford NER [9]; and *(iii)* QACM is a set of mappings that define the QA components in QAC in terms of the QA tasks that they implement. Mappings in QACM correspond to conjunctive rules, where the head is a predicate in QAC and the body is a conjunction of predicates in QAV. QA developer requirements are also represented as conjunctive queries over the predicates in QAC. Moreover, the problem of QA Component Composition corresponds to the enumeration of combinations of QA components that implement a QA developer requirement. In the following sections, QAESTRO and the problem of QA composition are described in detail.

3.1 The Question Answering Tasks

QA systems implement abstract QA tasks to answer questions posed by humans. QA tasks include question analysis, query construction, and the evaluation of the generated query over a knowledge base to answer the input question [22]. Figure 2 depicts an abstract pipeline of the QA tasks [22], which receives a question as input and outputs the answers to this question over a knowledge base.

Question Analysis: Using different techniques, the input question is analyzed linguistically to identify syntactic and semantic features. The following techniques form important subtasks:

Tokenization: A natural language question is fragmented into words, phrases, symbols, or other meaningful units known as tokens.

POS Tagging: The part of speech, such as noun, verb, adjective, and pronoun, of each question word is identified and attached to the word as a tag.

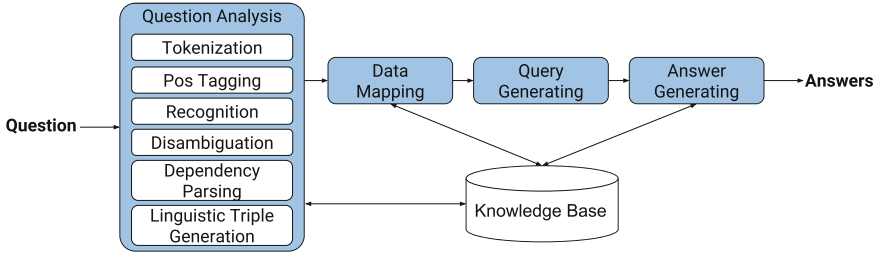


Fig. 2. Pipeline of QA Tasks. A QA pipeline receives a question and outputs the question answers. Question Analysis allows for question linguistic and semantic analysis to identify question features. During Data Mapping, question features are mapped into concepts in a Knowledge Base. A SPARQL query is constructed and executed during Question and Answer Generation.

Dependency Parsing: An alternative form of syntactic representation of the question to form a tree-like structure is created where arcs (edges in the tree) indicate that there is a grammatical relation between two words, whereas the nodes in the tree are the words (or tokens) in the question.

Recognition: An input question is parsed to identify the sequence of words that represent a person, a thing, or any other entity.

Disambiguation: The identity of the entity in the text is retrieved and then linked to its mentions in knowledge bases. Input for this may be one or more of the following: question, entity, and template. The output is a list of disambiguated entities.

Linguistic Triple Generation: Based on the input natural language question, triple patterns of the form $\langle \text{query term}, \text{relation}, \text{term} \rangle$ are generated [22].

Data Mapping: Information generated by Query Analyzer such as entities and tokens is mapped to its mentions in online knowledge bases such as DBpedia.

Query Generating: SPARQL queries are constructed; generated queries represent input questions over entities and predicates in online knowledge bases.

Answer Generating: The SPARQL queries are executed on the end points of knowledge bases to obtain the final answer.

Other QA Tasks include:

- **Question Type Identification:** This task identifies the type of the question. The input is the natural language question; the output is the type of the question, e.g., “yes-no”, “location”, “person”, “time”, or “reason”.
- **Answer Type Identification:** This task identifies the desired type of the final answer. This task is sometimes performed as a part of the Question Analysis task or as a subtask of Answer Generation.
- **Query Ranking:** In some of the QA systems, the task Query Generation generates multiple candidate queries. This task ranks the generated SPARQL queries using a ranking function and it helps to select the best ranked query.

- **Syntactic Parsing:** The input question is represented in the form of a syntactic tree, consisting of identified nouns, verbs, adjectives, relations, etc. However, this task may use as input natural language question or POS tags which makes it different from POS Tagging [21].

The above QA task definitions describe the logical structure of an abstract QA pipeline. However, QA systems implement these tasks differently, sometimes combining several of these tasks in different order or skipping some of the tasks.

3.2 Controlled Vocabulary for Abstract Question Answering Tasks

A vocabulary QAV of the domain of QA tasks is described as a pair $\langle \delta, A \rangle$, where δ is a signature of a logical language and A is the set of axioms describing the relationships among vocabulary concepts. A signature δ is a set of predicate and constant symbols, from which logical formulas can be constructed, whereas the axioms A describe the vocabulary by illustrating the relationships between concepts. For instance, the term *disambiguation* is a predicate of arity four in δ ; $disambig(x, y, z, t)$ denotes that the QA task disambiguation relates an entity x , a question y , a disambiguated entity z , and a template t . Furthermore, the binary predicate $questionAnalysis(x, y)$ models the question analysis task and relates an entity x to a question y . The following axiom states that the disambiguation task is a subtask of the question analysis task:

$$disambig(x, y, z, t) \rightarrow questionAnalysis(x, y)$$

3.3 Semantic Descriptions of Question Answering Components

QAC is a set of predicate signatures $\{QAC_1, \dots, QAC_n\}$ that model QA components. For example, AGDISTIS [23] is represented with predicate $Agdistis(x, y, z)$ where x , y , and z denote an entity, a question, and a disambiguated entity, respectively. Further, the QA component Stanford NER [9] is modeled with the predicate $StanfordNER(y, x)$, which relates a question y to an entity x .

QAESTRO follows the Local-As-View (LAV) approach to define QA components in QAC based on predicates in QAV. LAV is commonly used by data integration systems to define semantic mappings between local schemas and views that describe integrated data sources and a unified ontology [20]. The LAV formulation allows QAESTRO to scale up to a large number of QA components, as well as to easily be adjusted to new QA components or modifications of existing ones. This property of the LAV approach is particularly important in the area of Question Answering, where new QA systems and components are constantly being proposed by practitioners and the research community. Following the LAV approach, a QA component C is defined using a conjunctive rule R . The head of R corresponds to the predicate in QAC that models C , while the body of R is a conjunction of predicates in QAV that represent the tasks performed by C . LAV rules are safe, i.e., all the variables in the head of a rule are also variables in the predicates in the body of the rule. Additionally, input and output restrictions

of the QA components can be represented in LAV rules. The following LAV rules illustrate the semantic description of the QA components AGDISTIS and Stanford NER in terms of predicates in QAV. The symbol “\$” denotes an input attribute of the corresponding QA component.

$$\begin{aligned} Agdistis(\$x, \$y, z) &:- disambig(x, y, z, t), entity(x), question(y), disEntity(z) \\ StanfordNER(\$y, x) &:- recognition(y, x), question(y), entity(x) \end{aligned}$$

These rules state the following properties of AGDISTIS and Stanford NER:

- (i) AGDISTIS implements the QA task of disambiguation; an entity and a question are received as input, and a disambiguated entity is produced as output;
- (ii) Stanford NER implements the QA task of recognition; it receives a question as input and outputs a recognized entity.

3.4 Question Answering Developer Requirements

A QA developer requirement expresses the QA tasks that are required to be implemented by compositions of existing QA components. QA developer requirements are represented as conjunctive rules, where the body of a rule is composed of a conjunction of QA tasks. Similarly as for LAV mapping rules, input and output conditions can be represented; the symbol “\$” denotes attributes assumed as input in the QA developer requirement. For instance, consider a developer who is interested in determining those compositions of QA components that, given a question q , perform entity recognition and disambiguation, and produce as output an entity e ; the question q will be given as input to the pipeline.

$$QADevReq(\$q, e) :- recognition(q, e), disambig(e, q, de, t)$$

Now, suppose another developer requires also to know the compositions of QA components able to perform the pipeline of entity recognition and disambiguation. However, given the question as input, she requires to check all the intermediate results produced during the execution of the two tasks. In this case, the body of the rule remains the same, while the head of the rule ($QADevReq$) includes *all* variables corresponding to the arguments of the disambiguation task.

$$QADevReq(\$q, e, de, t) :- recognition(q, e), disambig(e, q, de, t)$$

4 Composition of QA Components in Pipelines

In this section, we describe the QAESTRO solution to the problem of QA Component Composition. We then describe the QAESTRO architecture, and the main features of the QAESTRO components.

4.1 The Problem of QA Component Composition

As previously presented, QAESTRO provides a vocabulary QAV that formalizes QA tasks, and allows for the definition of QA components using LAV rules and QA developer requirements using conjunctive queries based on QAV. In this subsection, we will show how QAESTRO solves the problem of QA Component Composition, i.e., how different QA components are automatically composed for a given developer requirement based on LAV mappings that semantically describe existing QA components. Next, we illustrate the problem of QA Component Composition. Besides the descriptions of AGDISTIS and Stanford NER (Subsect. 3.3), we consider further semantic descriptions for DBpedia NER [18], Alchemy API, and the answer type generator component of the QAKiS QA system [6], which we call *Qakisatype*.

$DBpediaNER(\$y, x) :- recognition(y, x), question(y), entity(x)$

$Alchemy(\$y, z) :- disambig(x, y, z, t), question(y), disEntity(z)$

$Qakisatype(\$y, a) :- answertype(y, a, o), question(y), atype(a)$

These rules state the following properties of the described QA components:

- The predicates $DBpediaNER(\$y, x)$, $Alchemy(\$y, z)$, and $Qakisatype(\$y, a)$ represent the QA components DBpedia NER, the Alchemy API, and Qakisatype, respectively. The symbol $\$$ denotes the input restriction of these QA components, i.e., the three QA components receive a question as input. These predicates belong to QAC.
- The predicates $recognition(y, x)$, $disambig(x, y, z, t)$, and $answertype(y, a, o)$ model the QA tasks: entity recognition, disambiguation, and answer type identification, respectively. These predicates belong to the QAV.
- An input natural language question is modeled by the predicate $question(y)$, while $entity(x)$ represents a named entity identified in a question.
- The QAV predicates $disEntity(z)$ and $atype(a)$ model the QA tasks of generating disambiguated entities and answer type identification, respectively.
- The variables x , y , z , and a correspond to instances of predicates $entity$, $question$, $disEntity$, and $atype$, respectively. The variable o is not bound to any predicate because *Qakisatype* does not produce ontology concepts.

Additionally, consider the following QA developer requirement for QA component compositions in a pipeline of entity recognition, disambiguation, and answer type identification, which receives a question q and outputs an entity e .

$QADevReq(\$q, e) :- recognition(q, e), disambig(e, q, de, t), answertype(q, a, o)$

QAESTRO generates two QA compositions as solutions to the problem of QA Component Composition. These compositions correspond to the enumeration of those combinations of QA components that implement the pipeline of the QA tasks of recognition, disambiguation, and answer type identification. Further, each composition satisfies the input restrictions of each QA component.

$QADevReq(\$q, e) : - StanfordNER(\$q, e), Agdistis(\$e, \$q, de), Qakisatype(\$q, a)$ (1)

$QADevReq(\$q, e) : - DBpediaNER(\$q, e), Agdistis(\$e, \$q, de), Qakisatype(\$q, a)$ (2)

Composition (1) indicates that the combination of the QA components Stanford NER, AGDISTIS, and Qakisatype implements the pipeline of recognition, disambiguation, and answer type identification. The input restriction of $StanfordNER(\$q, e)$ is satisfied by the question that is given as input in the pipeline. The QA component $Agdistis(\$e, \$q, de)$ is next in the composition; both the entity e produced by Stanford NER and the question q given by input to the pipeline, satisfy the input restrictions of this QA component. Similarly, input restriction of $Qakisatype(\$q, a)$ is satisfied by the question q . Additionally, Composition (2) implements the pipeline, but the QA component DBpedia NER is utilized for the QA task of entity recognition. The input restriction of DBpedia NER is also satisfied by the question received as input of the pipeline.

Consider the following compositions for the same QA developer requirement:

$QADevReq(\$q, e) : - StanfordNER(\$q, e), Alchemy(\$q, de), Qakisatype(\$q, a)$ (3)

$QADevReq(\$q, e) : - DBpediaNER(\$q, e), Alchemy(\$q, de), Qakisatype(\$q, a)$ (4)

Both compositions implement the pipeline of recognition, disambiguation, and answer type identification; also the input restrictions of the QA components are satisfied. However, these compositions are not valid because the argument e that represents an entity is not generated by Alchemy. This argument is required to be joined with the entity produced by the QA component that implements the entity recognition task and to be output by the compositions.

Formally, the problem of QA Component Composition is cast to the problem of Query Rewriting using LAV views [2]. An instance of QRP receives a set of LAV rules on a set P of predicates that define sources in V , and a conjunctive query Q over predicates in P . The output of Q is the set of valid re-writings of Q on V . Valid rewritings QR of Q on V are composed of sources in V that meet the following conditions:

- Every source in QR implements at least one subgoal of Q .
- If S is a source in QR and implements the set of subgoals SG of Q , then
 - The variables in both the head Q and SG are also in the head of S .
 - The head of the LAV rule where S is defined, includes the variables in SG that are in other subgoals of Q .

Note that the QA component $Alchemy(q, de)$ violates these conditions in Composition (3) and (4), i.e., $Alchemy(q, de)$ does not produce an entity e for a question q . Thus, compositions that implement the QA task of disambiguation with $Alchemy(q, de)$ are not valid solutions for this QA developer requirement.

QAESTRO casts the problem of QA Component Composition into the Query Rewriting Problem (QRP). Deciding if a query rewriting is a solution of QRP is NP-complete in the worst case [20]. However, given the importance of QRP in data integration systems and query optimization, QRP has received a lot of

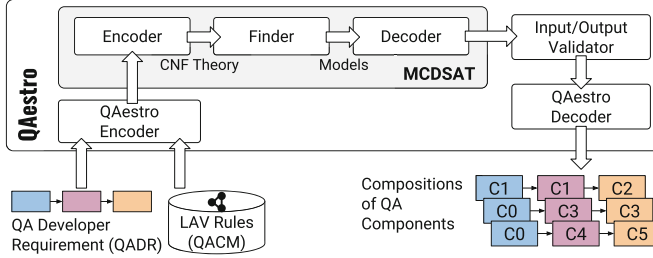


Fig. 3. QAESTRO Architecture. QAESTRO receives as input a QA developer requirement QADR and a set QACM of LAV rules describing QA components, and produces all the valid compositions that implement QADR.

attention in the Database area, and several approaches are able to provide effective and efficient solutions to the problem, e.g., MCDSAT [2, 13] or GQR [14]. Thus, building on existing solutions for QRP, we devise a solution to the problem of QA Component Composition that is able to efficiently and effectively enumerate valid compositions of a QA developer requirement. QAESTRO implements a two-fold approach, where first, solutions to the cast instance of QRP are enumerated. Then, input and output restrictions of QA components are validated. Valid compositions of QA components that both implement a QA developer requirement and respect the input and output restrictions, are produced as solutions of an instance of the problem of QA Component Composition.

4.2 The QAESTRO Architecture

QAESTRO relies on MCDSAT, a state-of-the-art solver of QRP to efficiently enumerate the compositions of QA components that correspond to implementations of a QA developer requirement. Figure 3 depicts the QAESTRO architecture. QAESTRO receives as input a QA developer requirement QADR expressed as a conjunctive query over QA tasks in a vocabulary QAV. Furthermore, a set QACM of LAV rules describing QA components in terms of QAV is given as input to QAESTRO. QACM and QADR correspond to an instance of the QA Component Composition which is *cast* into an instance of QRP and passed to MCDSAT, a solver of QRP. MCDSAT encodes the instance of QRP into a CNF theory in a way that *models* of this theory correspond to solutions of QRP. MCDSAT utilizes an off-the-shelf SAT solver to enumerate all *valid* query rewritings that correspond to *models* of the CNF theory. The output of the SAT solver is decoded, and input and output restrictions are validated in each query rewriting. Finally, QAESTRO decodes valid query rewritings where input and output restrictions are satisfied, and generates the compositions of QA components that implement the pipeline of QA tasks represented by QADR.

5 Empirical Study

We empirically study the behavior of QAESTRO in generating possible QA component compositions given QA developer requirements. We assess the following research questions: **(RQ1)** Given the formal descriptions of QA components using QAV and QA developer requirements are we able to produce sound and correct compositions? **(RQ2)** Are we able to produce efficiently solutions to the problem of QA Component Composition? The experimental configuration is as follows:

QA Components and Developer Requirements. To evaluate QAESTRO empirically, we have semantically described 51 QA components implemented by 20 QA systems which have participated in the first five editions of the QA over Linked Data Challenge (QALD1–5)⁴. Additionally, we studied well-known QA systems such as AskNow [7], TBSL [21], and OKBQA to semantically describe their components. After closely examining more than 50 components of these QA systems, we broadly categorized the components based on the QA tasks they perform, as defined in Sect. 3.1. For defining the LAV mappings, we selected only those QA components, for which there is a clear statement about input, output, and the QA tasks they perform in a publication (i.e., scientific paper, white paper, or source repository) about the respective QA system. Furthermore, we constructed manually 30 QA developer requirements for standalone QA tasks and QA pipelines integrating various numbers of QA tasks.

Metrics. (i) *Number of QA component compositions*: Number of QA component compositions given the semantic descriptions of QA components in QACM and a QA developer requirement; (ii) *Processing Time*: Elapsed time between the submission of a QA developer requirement and the arrival of all the QA component compositions produced by QAESTRO.

Implementation. QAESTRO is implemented in Python 2.7 on top of MCD-SAT [2], which solves QRP with the use of the off-the-shelf model compilation and enumeration tool c2d⁵. QAESTRO source code can be downloaded from <https://github.com/WDAqua/QAestro> and the evaluation results can be viewed at <https://wdaqua.github.io/QAestro/>. Experiments were executed on a laptop with Intel i7-4550U, 4 × 1.50 GHz and 8 GB RAM, running Fedora Linux 25.

5.1 Evaluation Results

Analysis of QA Components. In Fig. 4, we illustrate all QA components that have been formalized using QAESTRO along with their connections to the QA tasks they implement and the QA systems they belong to as an undirected graph⁶. In total, the resulting graph consists of 82 nodes and 102 edges. From the 82 nodes, 20 correspond to QA systems, 11 represent QA tasks, and 51 refer

⁴ <http://qald.sebastianwalter.org/>.

⁵ <http://reasoning.cs.ucla.edu/c2d/>.

⁶ The graph visualization was generated with cytoscape - <http://www.cytoscape.org>.

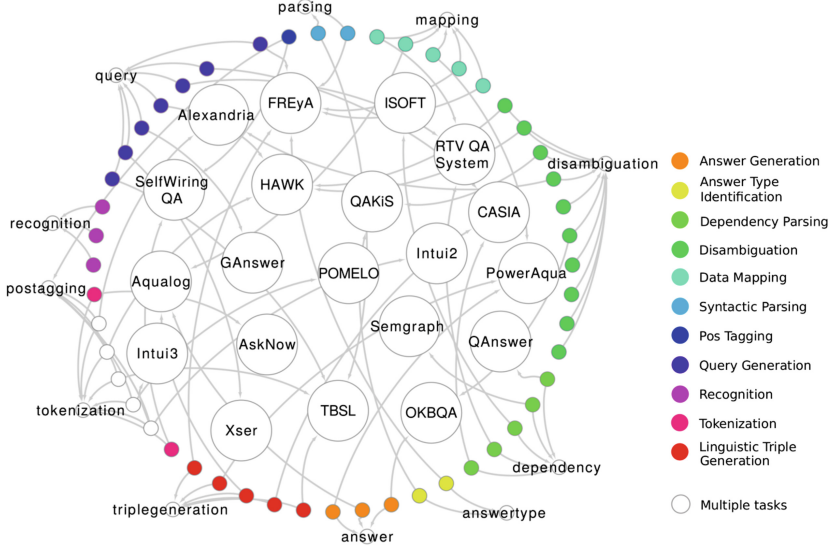


Fig. 4. QA Systems, Components, and Tasks. 51 QA components from 20 QA systems, implementing 11 distinct QA tasks are depicted as a directed graph.

to concrete QA components – 43 are part of the QA systems while 8 are provided also as standalone components (e.g., AGDISTIS, DBpedia NER, etc.). It can be observed in Fig. 5a that the majority of the analyzed QA components implement the Disambiguation task (10 in total) followed by the Query Generation (8), Tokenization (8), and POS Tagging (7) tasks. Many of these components are reused among the different QA systems. In addition, Fig. 5b shows that in almost half of the QA systems, components that implement Tokenization and Query Generation are included, while some less popular QA tasks like Answer Type Identification and Syntactic Parser are part of only two QA systems.

QA Component Compositions. In order to evaluate the efficiency of QAESTRO, we edited 30 QA developer requirements with different number of QA tasks to be included in the QA pipeline and different expected inputs and outputs. Given these requirements and the semantic descriptions of QA components QAESTRO produced possible QA component compositions. Figure 6a reports on the number of different compositions for all 30 requirements grouped according to the number of QA tasks they include. Figure 6b demonstrates the time needed by QAESTRO to process each of the requirements and generate QA component compositions. We performed the measurements 10 times and calculated the mean values. While for standalone QA components or components that perform two tasks the solution space is relatively limited – from one to 30 combinations – for QA developer requirements that include three or more QA tasks the number of QA compositions may increase significantly. For instance, we notice that for a few requirements with three and four QA tasks the possible compositions are

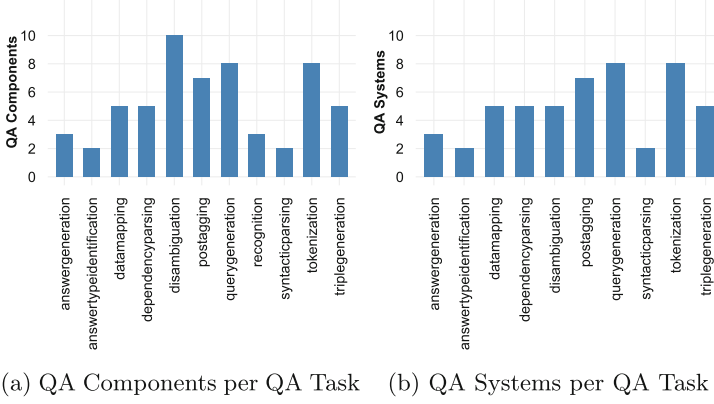


Fig. 5. Frequencies of QA components and Systems per QA Task. Disambiguation, Tokenization, and Query Generation are the most popular tasks.

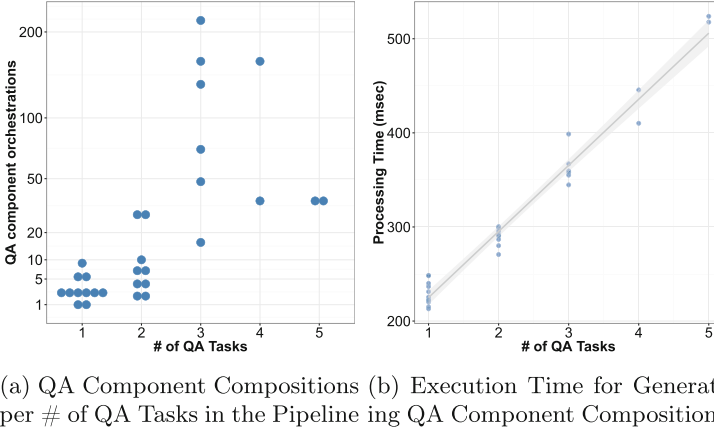


Fig. 6. Analysis of QA Component Compositions. QAESTRO is able to produce QA component compositions effectively and very fast.

more than 100. In these cases, the requirements do not foresee input or output dependencies between QA components, hence, the number of possible combinations increases significantly. All solutions produced by QAESTRO are sound and complete, since MCDSAT is able to produce every valid solution and all solutions that it provides are valid [2]. Furthermore, the processing time is for all requirements less than half a second and relates linearly to the number of QA tasks, since MCDSAT can perform model enumeration in linear time. Consequently, the experimental results allow us to positively answer **RQ1** and **RQ2**.

6 Related Work

Since 2010, more than 70 Question Answering Systems have been developed [12]. However, most of these QA systems (e.g., [6, 7]) are monolithic in their implementation, which restricts their reusability in other QA approaches. Therefore, researchers have shifted their focus to building reusable QA architectures and frameworks. QALL-ME [8] is one such framework; it provides a reusable architecture to build multilingual QA systems. Furthermore, openQA [17] is an extensible framework for building multiple QA pipelines. It includes many external QA systems such as TBSL [21] to build modular QA pipelines. Open Knowledge Base and Question Answering (OKBQA) is a recent attempt to build component-based QA systems. Its repository includes overall 24 QA components solving four core QA tasks. However, there is no formalized way to describe how standalone QA components like Stanford NER [9] and Alchemy API could be used to perform the disambiguation task in OKBQA pipeline besides existing OKBQA disambiguation components. In 1978, researchers first attempted to provide formalization for QA systems [4]. The authors illustrated how a natural language question can be translated into a semantic representation and an underlying knowledge system can be formally described. Qanary [5], co-developed by some authors of this paper, is a recent attempt to provide a formalised methodology for building vocabulary-driven QA systems by integrating reusable QA components. QAESTRO can be integrated into Qanary, and allow for the semantic description and automatic composition of QA components available in Qanary.

The problem of Web services selection and composition has been extensively studied in the literature (e.g., [3, 13]). Existing approaches range from heuristic-based [3] to SAT solver-based methods [13], and have shown to be efficient and effective for different instances of the problem. However, techniques proposed by Izquierdo et al. [13] that exploit the properties of SAT solvers, have provided evidence for large-scale composition of Web services. QAESTRO also exploits the benefits of modern SAT solvers and makes available an effective and efficient solution to the problem of QA Component Composition.

7 Conclusions and Future Work

In this work, we have tackled the problem of QA Component Composition by casting it to the Query Rewriting Problem. We introduced QAESTRO, a framework that enables QA developers to semantically describe QA components and developer requirements by exploiting the LAV approach. Moreover, QAESTRO computes compositions of QA components for a given QA developer requirement by taking advantage of SAT solvers. In an empirical evaluation, we tested QAESTRO with various QA developer requirements for QA pipelines of varying complexity, containing from two to five tasks. We observed that QAESTRO can not only produce sound and valid compositions of QA components, but also demonstrates efficient processing times. QAESTRO can successfully deal with the growing number of QA systems and standalone QA components, that is, the

appearance of a new QA component only causes the addition of a new mapping describing the QA component in terms of the concepts in the QA vocabulary. Automated composition of QA components will enable subsequent research towards determining and executing best-performing QA pipelines that achieve better performance in terms of accuracy (precision, recall) and execution time. Currently, QAESTRO is not capable of implementing the QA pipeline in an automated way to answer an input question, however, in the future, QAESTRO will be integrated in approaches like Qanary to automatically retrieve all feasible combinations of available QA components and to realize the best-performing QA pipeline in concrete use cases.

Acknowledgements. Parts of this work received funding from the EU Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 642795 (WDAqua).

References

1. Androutsopoulos, I., Ritchie, G.D., Thanisch, P.: Natural language interfaces to databases - an introduction. *Nat. Lang. Eng.* **1**(1), 29–81 (1995)
2. Arvelo, Y., Bonet, B., Vidal, M.: Compilation of query-rewriting problems into tractable fragments of propositional logic. In: *Proceedings of the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference* (2006)
3. Berardi, D., Cheikh, F., Giacomo, G.D., Patrizi, F.: Automatic service composition via simulation. *Int. J. Found. Comput. Sci.* **19**(2), 429–451 (2008)
4. Bolc, L. (ed.): *Natural Language Communication with Computers*. LNCS, vol. 63. Springer, Heidelberg (1978). doi:[10.1007/BFb0031367](https://doi.org/10.1007/BFb0031367)
5. Both, A., Diefenbach, D., Singh, K., Shekarpour, S., Cherix, D., Lange, C.: Qanary-a methodology for vocabulary-driven open question answering systems. In: *ESWC* (2016)
6. Cabrio, E., Cojan, J., Aprosio, A.P., Magnini, B., Lavelli, A., Gandon, F.: QAKiS: an open domain QA system based on relational patterns. In: *Proceedings of the ISWC 2012 Posters and Demonstrations Track* (2012)
7. Dubey, M., Dasgupta, S., Sharma, A., Höffner, K., Lehmann, J.: AskNow: a framework for natural language query formalization in SPARQL. In: Sack, H., Blomqvist, E., d’Aquin, M., Ghidini, C., Ponzetto, S.P., Lange, C. (eds.) *ESWC 2016*. LNCS, vol. 9678, pp. 300–316. Springer, Cham (2016). doi:[10.1007/978-3-319-34129-3_19](https://doi.org/10.1007/978-3-319-34129-3_19)
8. Ferrández, Ó., Spurk, C., Kouylekov, M., Dornescu, I., Ferrández, S., Negri, M., Izquierdo, R., Tomás, D., Orasan, C., Neumann, G., Magnini, B., González, J.L.V.: The QALL-ME framework: a specifiable-domain multilingual question answering architecture. *J. Web Semant.* **9**(2), 137–145 (2011)
9. Finkel, J.R., Grenager, T., Manning, C.D.: Incorporating non-local information into information extraction systems by Gibbs sampling. In: *43rd Annual Meeting of the Association for Computational Linguistics ACL* (2005)
10. Gomes, C.P., Kautz, H.A., Sabharwal, A., Selman, B.: *Satisfiability Solvers* (2008)
11. Halevy, A.Y.: Answering queries using views: a survey. *VLDB J.* **10**(4), 270–294 (2001)

12. Höffner, K., Walter, S., Marx, E., Usbeck, R., Lehmann, J., Ngonga Ngomo, A.-C.: Survey on challenges of question answering in the semantic web. *Semant. Web J.* (2016). <http://www.semantic-web-journal.net/content/survey-challenges-question-answering-semantic-web>
13. Izquierdo, D., Vidal, M.-E., Bonet, B.: An expressive and efficient solution to the service selection problem. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) *ISWC 2010. LNCS*, vol. 6496, pp. 386–401. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-17746-0_25](https://doi.org/10.1007/978-3-642-17746-0_25)
14. Konstantinidis, G., Ambite, J.L.: Scalable query rewriting: a graph-based approach. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2011)
15. Levy, A.Y., Rajaraman, A., Ordille, J.J.: Querying heterogeneous information sources using source descriptions. In: *Proceedings of 22th International Conference on Very Large Data Bases* (1996)
16. López, V., Uren, V.S., Sabou, M., Motta, E.: Is question answering fit for the semantic web?: a survey. *Semant. Web* **2**(2), 125–155 (2011)
17. Marx, E., Usbeck, R., Ngomo, A.N., Höffner, K., Lehmann, J., Auer, S.: Towards an open question answering architecture. In: *SEMANTICS* (2014)
18. Mendes, P.N., Jakob, M., García-Silva, A., Bizer, C.: DBpedia spotlight: shedding light on the web of documents. In: *Proceedings of the 7th International Conference on Semantic Systems, I-SEMANTICS* (2011)
19. Singh, K., Both, A., Diefenbach, D., Shekarpour, S.: Towards a message-driven vocabulary for promoting the interoperability of question answering systems. In: *ICSC* (2016)
20. Ullman, J.D.: Information integration using logical views. *Theor. Comput. Sci.* **239**(2), 189–210 (2000)
21. Unger, C., Bühmann, L., Lehmann, J., Ngomo, A.N., Gerber, D., Cimiano, P.: Template-based question answering over RDF data. In: *WWW* (2012)
22. Unger, C., Freitas, A., Cimiano, P.: An introduction to question answering over linked data. In: Koubarakis, M., Stamou, G., Stoilos, G., Horrocks, I., Kolaitis, P., Lausen, G., Weikum, G. (eds.) *Reasoning Web 2014. LNCS*, vol. 8714, pp. 100–140. Springer, Cham (2014). doi:[10.1007/978-3-319-10587-1_2](https://doi.org/10.1007/978-3-319-10587-1_2)
23. Usbeck, R., Ngonga Ngomo, A.-C., Röder, M., Gerber, D., Coelho, S.A., Auer, S., Both, A.: AGDISTIS - graph-based disambiguation of named entities using linked data. In: Mika, P., et al. (eds.) *ISWC 2014. LNCS*, vol. 8796, pp. 457–471. Springer, Cham (2014). doi:[10.1007/978-3-319-11964-9_29](https://doi.org/10.1007/978-3-319-11964-9_29)

Database and Expert Systems Applications

28th International Conference, DEXA 2017, Lyon,

France, August 28-31, 2017, Proceedings, Part I

Benslimane, D.; Damiani, E.; Grosky, W.I.; Hameurlain,

A.; Sheth, A.; Wagner, R.R. (Eds.)

2017, XXIV, 517 p. 172 illus., Softcover

ISBN: 978-3-319-64467-7