WP3-D2

Key Query Generation and Answering Engine

Project full title:	Knowledge Driven Data Exploitation
Project acronym:	K-Drive
Grant agreement no.:	286348
Project instrument:	EU FP7/Maria-Curie IAPP/PEOPLE WP 2011
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	UNIABDN, IBM/WP3-D2/D/PU/a1
Responsible editors:	Honghan Wu, Yuan Ren, Man Zhu, Jeff Pan
Reviewers:	Yuting Zhao
Contributing participants:	UNIABDN, IBM
Contributing workpackages:	WP3
Contractual date of deliverable:	31 December 2013
Actual submission date:	-

Abstract
Due to the increasing volume of and interconnections between semantic datasets, either in the linked open data cloud or within individual organisations, it becomes a challenging task for non-expert users to know what are included in a dataset and how they can make use of them. There has been abundant work on answering semantic queries, but little attention has been paid to the generation of queries with interesting results. In order to help novice users exploring semantic datasets, in this deliverable we follow the conceptual framework of insightful queries defined in deliverable 3.1 and propose an implementation framework to automatically generate such queries, identify their relations and produce relevant results. To verify our approach, we implemented our framework in a graph-pattern based approach and evaluated its performance with benchmark and real world datasets.

Keyword List
query generation, query answering, SPARQL querying, data understanding

Project funded by the European Commission within the 7th Framework Programme/Maria Curie Industry-Academia Partnerships and Pathways schema/PEOPLE Work Programme 2011.

© *K-Drive 2013.*

Key Query Generation and Answering Engine

Honghan Wu[1], **Yuan Ren**[1], **Man Zhu**[2] **and Jeff Pan**[1]

[1]Department of Computing Science, Aberdeen University, UK
[2]School of Computer Science and Technology, Southeast University, CN
Email: honghan.wu@abdn.ac.uk, y.ren@abdn.ac.uk,
zhuman.private@gmail.com and jeff.z.pan@abdn.ac.uk

-

Abstract
Due to the increasing volume of and interconnections between semantic datasets, either in the linked open data cloud or within individual organisations, it becomes a challenging task for non-expert users to know what are included in a dataset and how they can make use of them. There has been abundant work on answering semantic queries, but little attention has been paid to the generation of queries with interesting results. In order to help novice users exploring semantic datasets, in this deliverable we follow the conceptual framework of insightful queries defined in deliverable 3.1 and propose an implementation framework to automatically generate such queries, identify their relations and produce relevant results. To verify our approach, we implemented our framework in a graph-pattern based approach and evaluated its performance with benchmark and real world datasets.

Keyword List
query generation, query answering, SPARQL querying, data understanding

Contents

1 Introduction

In Deliverable 3.1, we investigated query generation problems in several domains. The main output of that effort is a query generation framework which illustrates the conceptual roadmap about how to meet users' information requirement when they are facing an unfamiliar dataset. Following such direction, in Deliverable 3.2, we will investigate the automatic query generation method and also efficient query answering approach so that the designed goals of work package 3 can be realised.

One of the main challenges for exploiting semantic datasets is to help users to understand the usefulness of the data, in terms of what kinds of queries can be answered with the given semantic datasets, without requiring users to be aware of the complexity of the underlying data model.

Query answering is one of the most widely studied and used services provided by semantic knowledge bases [Kollia et al., 2011]. Whereas query generation is also an important and non-trivial job to help users exploring a semantic data set. This is mainly due to the following two reasons:

1. **Constructing a good query is not easy for many users.**

   There can be different kinds insightful queries, such as *queries with the same set of answers*. Here we give some examples against some existing data sets to illustrate one kind of such queries — queries

   For example, in the BBC-PEEL dataset,[1] the following two queries Q1 and Q2 return the same results, suggesting that all musical work has a performance:

```
#Q1: Return all MusicalWork
SELECT ?m WHERE {
  ?m rdf:type dbtune:MusicalWork . }
#Q2: Return all those that have a Performance
SELECT ?y WHERE {
  ?x dbtune:performance_of ?y .
  ?x rdf:type dbtune:Performance .  }
```

   For example, in the Lehigh University Benchmark (LUBM),[2] the following queries Q1 and Q2 have the same results under RDF simple interpretation (i.e. without reasoning [Heino and Pan., , Fokoue et al., 2012]).

```
#Q1: Return those who take a Course
SELECT ?x WHERE {
  ?x lubm:takeCourse ?y .
  ?y rdf:type lubm:Course .  }
#Q2: Return Undergraduates who take a Course
SELECT ?y WHERE {
  ?x lubm:takeCourse ?y .
  ?x rdf:type lubm:Undergraduate .
  ?y rdf:type lubm:Course .  }
```

   This implies that only undergraduate students are taking courses in the dataset, which might be somehow *surprising*, as post-graduate students are *not* taking any courses. This gives users a sense of the quality of the query answers when reasoning is disabled.

[1] http://datahub.io/dataset/dbtune-john-peel-sessions

[2] http://swat.cse.lehigh.edu/projects/lubm/

Query generation (QG) has been studied in the field of database, based on database schemas (e.g. [Slutz, 1998]) or actual data (e.g. [Mishra et al., 2008]), where the main motivation is to generate queries for testing databases. A related research problem is query recommendation (QR), where query logs are widely used to generate queries based on querying and browsing behaviours of users [Chatzopoulou et al., 2009, Zhang and Nasraoui, 2006]. In short, the motivation of the database works on QR is closer to what we need but these works are mainly based on query logs. In this paper, we investigate the problem query generation for semantic data by analysing the data itself, rather than query logs.

[Guo et al., 2005]) ontology, which provides descriptions of terms such as *UndergraduateStudent*, *Course*, etc. However, administrators of universities are usually not technical experts, and it is not practical for them to go through the entire dataset to find interesting queries to ask. Such situations get even worse when multiple university datasets are inter-connected for purposes such as joint programme or credit transferring because users have to face much more data that they'ven never see before.

To overcome the above difficulties we believe it is important to develope technologies that automatically generate queries for users. Furthmore, such technical obstract usually means that we do not have previous user queries or user-defined profiles. Hence in this paper we will focus on generating queries purely from data sets, and leave query genration with user interference to future work.

2. **Queries can provide complementary knowledge to those asserted in the datasets.** Today's ontology languages are mainly based on the crisp description logics. Therefore, non-crisp knowledge such as trends, exceptions can not be easily represented with standard ontology formalisms such as OWL (Web Ontology Language).

   For example, in a disease dataset doctors might want to claim that a person's chance of having diabetes is positively correlated to its bodyweight. Such information usually requires probabilistic extensions of ontology languages to describe. While with top-k queries, users can realise such trending when they query for people with high bodyweight and people with diabetes. For example, in a university dataset it is very likely that every professor has a PhD degree. However due to possible exceptions, such knowledge should not be directly asserted in ontology. If a query for instances of *Professor* and a query for individuals who have *PhD Degree* are generated, users can realise such non-crisp knowledge when they observe that the solutions to both queries have a large overlapping. While the a few exceptions will be very interesting to users as well.

   Besides, as Motik et al. have pointed out [Motik et al., 2008], structured objects are difficult to be represented in OWL. To support the representation and reasoning of structured objects, they have extended the syntax of OWL and the algorithm of reasoners. Nevertheless, this futher increases the difficulty of modelling and understanding semantic datasets.Alternatively, queries that explicitly describe the structural relations between domain entities can be used to retrieve structured objects for users.

   From this perspective, query generation is essentially a process of extracting graph patterns that can not be easily represented with ontology axioms, evaluating their importance and identifying their correspondences.

Similar to QG for databases, there exists work on QG for testing semantic web engines [Grau and Stoilos, 2011, Görlitz et al., 2012], based on ontologies or parameterisations. d'Aquin and Motta [d'Aquin and Motta, 2011]

proposed a QG approach based on formal concept analysis, which uses computationally complex ontological reasoning. In this paper, we propose a tractable query generation approach based on data summarisation and graph patterns.

The following sections of this deliverable are organised as follows. Section 2 will give a framework of the proposed query generation and query answering prototype. In section 3, the key concept notions of entity graph pattern is introduced. Based on this notion, the EDP graph, the key enabling technique, is defined and its properties are given in terms of theorems. In section 4 and section 5 we introduce the two main tasks of query generation and query answering respectively. In Section 6, we give the evaluation results on real work scenarios and datasets. Finally, we conclude the work of this deliverable.

2 Query Generation and Answering Prototype

Figure 1gives the architecture of our query generation and answering prototype. The default input of the system is a dataset, and the default outputs are key queries and their results. In addition to the default I/O, this system can also function as a normal query/answering engine, which means that users can submit queries to get results. The main difference of this Query-Answering system is that it utilizes the pre-computed key query results to speed up user query execution. As shown in the figure, the system is composed of three main components. The first component is dataset summarizer which generates an EDP graph as the summarization of given dataset. In addition, it can also extract the T-Box from the dataset if there is such information in the given dataset. The EDP graph is a key intermediate result which is used as a guidance by query generation methods in the second component i.e. 'Syntactic Query Generator'. The summarization and the T-Box are both used as inputs for the third component 'Semantic Query Answering Engine'.

The 'Syntactic Query Generator' generates key queries to reveal the insights of given dataset for helping people understanding unfamiliar datasets. We propose three query generation methods each of which applies a different strategy for defining and locating the 'insights' of datasets. The 'baseline QG' is to locate 'important knowledge' based on metrics from [1] and [2]. 'Data mining QG' is based on dependencies of different queries. Queries with both high dependencies and low dependencies are mined out as insightful queries. The last method, 'guided QG', simply uses the information in the EDP graph by locating the important nodes and paths which connecting the important nodes. The importance is assessed based on the number of instances in the dataset.

The F1.2 component is called 'Semantic Query Answering' component. This component has an initialization step to populate necessary storages like indexes needed to serve the users as a query answering engine. The input of the initialization process are EDP graph, T-Box(optional) and Key Queries generated by the second component. For each query, the process will first decompose the queries into a set of atomic queries and then call a 3rd party RDF store to get results for all atomic queries. These results will be cached and reorganized in a dedicated structure called 'Indexed Key Query Result'. After this process, the component will be ready for accepting user queries. When a query comes, the component will call the first subcomponent of 'Query Dependency Resolver' to decompose the query into two parts. The first part is a set of atomic queries and the second part is irresolvable part. Then, the decomposed queries are sent to the next subcomponent 'Query Result Compositor'. This subcomponent will execute all sub-queries separately. After all results of sub-queries are collected, it will compose the result and return the result to the upper layer. The sub-query execution is served by the 'Query Executor', which will call one of the lower query-answering components according to the type of each sub-query.

The last component is 'Intelligent UI' which help users make use of data in a more user-friendly manner. The objective is to design and develop UI that helps users to understand and exploit the knowl-

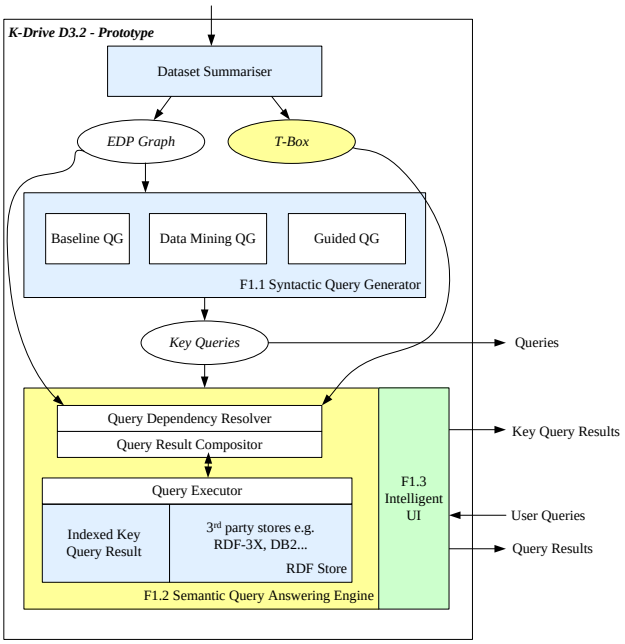

Figure 1: The Architecture of Query Generation and Answering Prototype

edge base based on the user modeling.

3 Graph Pattern Extraction

The graph pattern extraction module is responsible for providing data and service which are needed by following modules to do interesting graph pattern mining. The first outcome is a concise representation of the dataset in question. This representation is based on an atomic graph pattern i.e. entity description pattern which is also called as EDP for short. This representation is discussed in detail in the first subsection. The second output is a searching service which is the primary interface for the following components to do graph pattern mining. This searching service is based on several primitive EDP operations which are introduced in the second subsection. The implementation of the searching service is given in the last subsection.

3.1 Extracted Graph Pattern Model

Given an RDF graph, the graph pattern model is to generate a condensed description which can facilitate the graph pattern mining. In terms of structure, simplest graph patterns are those which only include one node. We can call these patterns as atomic patterns because it is impossible to divide them into smaller structures. By linking atomic patterns, one can construct more complex graph patterns. Following this idea, our extraction method applies a bottom-up strategy to summarize a semantic web dataset. Specifically, we propose an atomic pattern concept in which only one node is involved. Based on this

concept, we summarize the given RDF dataset as a new graph which describes the relations between atomic graph patterns.

3.2 Entity Description Pattern

A semantic web dataset is essentially an RDF graph. In such a graph, we call its non-literal nodes as entities. For each entity e in an RDF graph G, we can extract its describing data block by extract triples in G each of which has e as its subject or object. Formally, each entity e has an entity description block (EDB for short) as defined in Definition 1.

Definition 1 *(Entity Description Block)* $\forall e \in G$*, its description block is* $B_e = \{< e, p_i, o_i > | < e, p_i, o_i > \in G\} \cup \{< s_i, p_i, e > | < s_i, p_i, e > \in G\}$ *, where s and p are resources in G.*

For an entity description block, its summarization is introduced as a notion of entity description pattern (Definition 2). EDP, the short name for entity description pattern, is the atomic graph pattern in our summarization model.

Definition 2 *(Entity Description Pattern) Given an entity description block* B_e*, its description pattern is a tuple* $P_e = (C_e, A_e, R_e, V_e)$ *where*
$C_e = \{c_i | < e, rdf : type, c_i > \in G\}$ *is called as the class component ,*
$A_e = \{p_i | < e, p_i, l_i > \in G,\ l_i$ *is a literal*$\}$ *is called as the attribute component ,*
$R_e = \{r_i | < e, r_i, o_i > \in G,\ o_i$ *is a URI resource or blank node*$\}$ *is called as the relation component and*
$V_e = \{v_i | < s_i, v_i, e > \in G\}$ *is called as the reverse relation component.*

Essentially, an RDF graph G is a set of EDB i.e. $G = \cup_(e \in G) B_e$. By summarizing all entity description blocks in G, we can get the initial summarization result of G i.e. $\cup_(e \in G) P_e$. Given this initial result, we define a merge operation on EDPs which can further condense the summarization. Definition 3 defines the merge operation on EDPs which share the same class component. Based on this definition, we can merge any EDP set by grouping the EDPs in advance. Specifically, before merging, EDPs are grouped into a set of subsets according to their class components i.e. each subset contains a set of EDPs whose class components are identical and EDPs in different subsets have different class components. Then each of these subsets is merged into one EDP according to Definition 3. Finally, all merged EDPs are put together as the merge result.

Definition 3 *(EDP Merge) Given a set of EDPs* $\{P_i\}_{(i=1..n)}$ *whose elements have identical class component C, we can merge these EDPs into one result EDP as follows:* $Merge(\{P_i\}_{(i=1..n)}) = (C, \cup_{(\{A_i = Attr(P_i)\})} A_i, \cup_{(\{R_i = Rel(P_i)\})} R_i, \cup_{(\{V_i = Rev(P_i)\})} V_i)$*, where*
$Attr(P_i)$ *denotes the attribute component of* P_i*,*
$Rel(P_i)$ *denotes the relation component of* P_i*, and*
$Rev(P_i)$ *denotes the reverse relation component of* P_i*.*

The rationale behind this merge operation is that entities of the same type(s) might be viewed as a set of homogeneous things. This viewpoint is common in knowledge representation e.g. a class is viewed the set of all its individuals. Hence, in terms of graph pattern, the EDPs of all entities sharing the same type(s) should be merged into an integrated pattern. So far, we can define the EDP summarization of an RDF graph.

Definition 4 *(EDP Summarization) Given an RDF graph G, its EDP summarisation is defined by the following equation.* $Summ_{EDP(G)} = Merge(\cup_{(e \in G)} P_e)$

3.3 Entity Description Pattern Graph

EDP is the atomic graph pattern. Generally speaking, interesting queries usually correspond to more complex graph patterns. Hence, it would be more beneficial to know how EDPs are connected to each other in the original RDF graph.

To reveal more insights about the RDF dataset in question, we also introduce an EDP graph for characterize the linking structures in the original RDF graph.

Definition 5 *(EDP Graph) Given an RDF graph G, its EDP graph is defined as follows* $\mathcal{G}_{EDP}(G) = \{< P_i, l, P_j > | \exists e_i \in E(P_i), \exists e_j \in E(P_j), < e_i, l, e_j > \in G, P_i \in Summ_{EDP}(G), P_j \in Summ_{EDP}(G)\}$ *where* $E(P_i)$ *denotes the set of entities conforms to the EDP* P_i*. If* P_i *is not merged EDP,* $E(P_i)$ *is the set of entities from which* P_i *can be generated; if* P_i *is a merged one,* $(P_i) = \cup_{(P_k \in P)} E(P_k)$ *,* P *is the set of EDPs from which* P_i *is merged.*

As defined in Definition 5, EDP graph defines the pair-wise link relations between EDPs. From the definition, it can be figured out that if there is a link between entities of a pair of EDPs, they will have a link in the EDP graph. The EDP graph can be used to generate graph patterns or queries. Specifically, a sub-graph of the EDP graph can be converted into a query. It should be noted that if the sub-graph contains more than one links, the generated query might encounter empty result set on the original RDF graph. For example, if in the EDP graph we have $< Student, advisor, FullProfessor >$ and $< FullProfessor, headOf, Department >$, it is possible that in the RDF graph, the department head doesn't advise any student. However, it can be easily proved that EDP graph has a very good property as described in Theorem 1.

Theorem 1 *If a query Q has non-empty result set on RDF graph G, then its corresponding EDP graph* $i.e. \mathcal{G}_{EDP(Q)}$*, is a sub-graph of* $\mathcal{G}_{EDP(G)}$*.*

According to our definition, interesting queries are graph patterns hidden in the data. This means that interesting queries are non-empty queries. Combined this with the fact stated in Theorem 1, one can conclude that interesting queries have corresponding EDP structures which are sub-graphs of the EDP graph defined in Definition 5. Given all these propositions, we can conclude a fact stated in Lemma 1.

Lemma 1 *Given an RDF graph G, mining interesting queries from it is the task to find corresponding sub-graphs from the EDP graph* $\mathcal{G}_{EDP(G)}$*.*

To conclude, given an RDF graph G, the summarization module provides two kinds of condensed data for supporting later graph pattern mining, i.e. $Summ_{EDP(G)}$ and $\mathcal{G}_{EDP(G)}$. In addition, Lemma 1 gives us a guideline for how and where to do interesting graph pattern mining.

3.4 Primitive EDP Query Operations and Their Linkages

In addition to the summarized data, the mining process also needs to do some query execution like getting statistics of some graph patterns or even getting query results for graph patterns. To meet this requirement, we define several primitive EDP query operations which can be used to do query answering.

Definition 6 *(EDP Query) An EDP query Q is a tuple* $(C, A, \{< r, C_r >\}, \{< v, C_v >\})$*, where*
$\mathfrak{C}$ *is a set of URI resources denoting classes,*
$\mathfrak{A}$ *is a set of URI resources denoting attributes,*
$\mathfrak{r}$ *is a URI resource denoting a relation and* $\mathfrak{C}_r$ *is the class component of a EDP which is linked by current variable through r,*
$\mathfrak{v}$ *is a URI resource denoting a reverse relation and* $\mathfrak{C}_v$ *is the class component of a EDP which is linked to current variable through v.*

Similar to EDP being the atomic graph pattern, EDP query (defined in Definition 6) is the atomic query operation. According to its definition, an EDP query can express a graph structure whose diameter is up to 2 steps. The query results can either be a statistics result describing the number of results or the list of entities which matches the query.

To construct a query whose diameter is longer than 2, one will have to link two or more EDP queries. The EDP query linkage operation is defined in Definition 7.

Definition 7 *(EDP Query Linkage) Given two EDP queires* Q_1*,* Q_2 *and a relation r, the linkage is a triple* $< Q_1, r, Q_2 >$*.*

4 Query Generation Implementation

4.1 Candidate Insightful Queries

Given a target graph $\mathcal{D}$, the generation of *candidate insightful queries* for $\mathcal{D}$ can be regarded as a process of identifying typical graph patterns (or typical graph pattern pairs) having instances within $\mathcal{D}$, such that these typical graph patterns (or typical graph pattern pairs) provide users some insights about the structure of $\mathcal{D}$.

4.2 Typical Graph Patterns

Typical graph patterns are concerned with the structured relations among domain objects. While schema graphs (or ontologies) specify some global structure, typical instance graph patterns inform users some possibly additional structure in the current version of the graph. There can be different kinds of typical graph patterns, such as star-shaped graphs, shallow tree shaped graphs, deep tree shaped graphs and graphs with loops.

Definition 8 ***(Looped Graph Pattern)*** *A graph pattern is a* looped graph pattern *if it contains a circle of nodes.*

A looped graph pattern is a variable-looped graph pattern *if it contains a circle of variables.*

A query is a (variable-)looped query *if its corresponding graph pattern is a (variable-)looped graph pattern.*

In this paper, we are particularly interested in graphs with looped graph pattern, since loops reveal the multiplicity of the connections between objects, i.e. objects are connected in the dataset via multiple paths. While nominal-free[3] ontologies are not sufficiently expressive to accurately represent loops. In other words, graphs with loops might give users some insights on complex relations among objects that are not captured in the corresponding ontology.

[3]Nominal is one of the OWL features that could introduce scalability problem for reasoning.

In what follows, we first introduce the notion of graph pattern correspondence and then will revisit looped queries.

4.3 Graph Pattern Correspondence

Graph pattern correspondence is concerned with the relationships between two group of objects. In the Introduction, we gave some example of queries with the same set of answers. They can be formally described by the correspondence of two graph patterns as defined below:

Definition 9 ***(Graph Pattern Correspondence)*** *Given an RDF instance graph $\mathcal{D}$, two graph patterns GP_1 and GP_2 correspond on a vector of variables v IFF there is a variable-free substitution $v \rightarrow v'$ such that $GP_{1(v\rightarrow v')} \in I_{v,\mathcal{D}}(GP_1)$ and $GP_{2(v\rightarrow v')} \in I_{v,\mathcal{D}}(GP_2)$. v' is called the v-correspondence of GP_1 and GP_2 w.r.t. $\mathcal{D}$.*

We use $C_{\mathcal{D},v}(GP_1, GP_2) = \{v' | v'$ is a v-correspondence of GP_1 and GP_2 w.r.t. $\mathcal{D}\}$ to denote the set of all v-correspondences. In the rest of the paper, we omit $\mathcal{D}$ from the notations when it is clear from context.

From the above definition, it is obvious that two graph patterns GP_1 and GP_2 correspond on a vector of variables v IFF there is a solution to $Q(GP_1)$ and a solution to $Q(GP_2)$ that have the same value assigned to v. While $C_{\mathcal{D},v}(GP_1, GP_2)$ actually indicates the different values of v that can be shared by solutions of $Q(GP_1)$ and $Q(GP_2)$. The following theorem shows the relation between graph pattern correspondence and conjunctive query answering:

Theorem 2 *Two graph patterns GP_1 and GP_2 corresponds on variables v IFF $Q(GP_1) \wedge Q(GP_2)$ has a solution.*

$C_{\mathcal{D},v}(GP_1, GP_2) = \{v' | v'$ is the value assigned to v in some solution of $Q(GP_1) \wedge Q(GP_2)$ w.r.t. $\mathcal{D}\}$.

This result can be utilised to generate the following kinds of insightful queries.

1. Queries with strong Correspondence: For two graph patterns GP_1 and GP_2, let v be a vector of all their shared variables, $Q(GP_1)$ and $Q(GP_2)$ are insightful queries with strong correspondence if $|C_v(GP_1, GP_2)|$ is higher or lower enough w.r.t. $|I_v(GP_1)|$ or $|I_v(GP_2)|$, which indicates that $Q(GP_1)$ and $Q(GP_2)$ share a lot, or very few solutions on variables in v, respectively.

This is because, $|I_v(GP_1)|$ and $|I_v(GP_2)|$ are the number of different solutions assigned to v in $Q(GP_1)$ and $Q(GP_2)$, respectively, and $|C_v(GP_1, GP_2)|$ is the number of different solutions assigned to v in both $Q(GP_1)$ and $Q(GP_2)$. When $\frac{|C_v(GP_1,GP_2)|}{I_v(GP_1)}$ is close to 1, it indicates that most of the solutions to $Q(GP_1)$ can also be regarded as solutions to $Q(GP_2)$ for all variables they share. When it is close to 0, it indicates that only very few solutions to $Q(GP_1)$ can be regarded as solutions to $Q(GP_2)$. Both queries with high shared solutions and the queries with low shared solutions as insightful queries.

Note that such a solution-sharing relation is not symmetric, i.e. it is possible that $Q(GP_1)$ shares many solutions with $Q(GP_2)$ but $Q(GP_2)$ only shares a few with $Q(GP_1)$.

2. Queries on Exceptions: With the correspondence defined in Definition 9 we can generate insightful queries due to exceptions.

- For a pair of queries Q_1 and Q_2 with very high correspondence, $Q_1 \wedge \{$`FILTER NOT EXISTS` $(Q_2)\}$ (or $Q_2 \wedge \{$`FILTER NOT EXISTS` $(Q_1)\}$) will be an insightful query on exceptions. Obviously, if two queries share a lot solution, then the solutions that do not belong to both of them are quite interesting to users.

- For a pair of queries Q_1 and Q_2 with strong low correspondence, $Q_1 \wedge Q_2$ will also be an insightful query on exceptions. If two queries share very few solutions, then the solutions that belong to both of them are interesting to users.

An extreme case of queries on exception is empty queries. In this paper, an empty query is a query that does not have any solution on the given RDF dataset, rather than arbitrary datasets [Baader et al., 2010], in which a query is empty if it does not have any solution for any dataset. Our notion of emptiness is based on the input instance graph(s), while their notion of emptiness is forced by the input ontology (schema graph). Our notion of empty queries is weaker and cannot be checked by their approach.

Now that we introduce the notion of graph pattern correspondence. Let us revisit looped queries, some of which can be regarded as a special extension of queries with high correspondence: for two graph pattern GP_1 and GP_2, if there is a path of variables $v_1, v_2, \ldots, v_n$ in GP_1 and a path of variables $u_1, u_2, \ldots, u_m$ in GP_2, $v_1 = u_1$ and $v_n = u_m$ are variables shared by GP_1 and GP_2, and GP_1 and GP_2 have high correspondence w.r.t. vector $\langle v_1, v_n \rangle$, then $Q(GP_1) \wedge Q(GP_2)$ is a looped query.

This is obvious, since by combining GP_1 and GP_2 we have a looped graph pattern containing variable loop v_1, v_2, $\ldots$, v_n, u_{m-1}, $\ldots$, u_2, v_1. And this looped graph pattern can be transformed to $Q(GP_1) \wedge Q(GP_2)$. If GP_1 and GP_2 have high correspondence, it indicates that the solutions to $Q(GP_1)$ construct loop structures with solutions to $Q(GP_2)$. Such loop structures are captured by solutions of the looped query $Q(GP_1) \wedge Q(GP_2)$.

4.4 Query Generation Framework

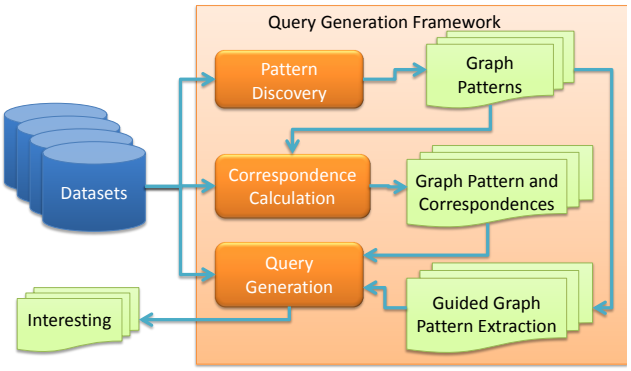

Figure 2: Query Generation Framework.

Our framework is depicted in Figure. 2. The first step identifies the graph patterns in datasets and extract their corresponding instances. The input of this step include the datasets and optionally some related constraints, such as size on the graph patterns. The output of this step is a set of pairs $\langle GP, I_{\mathcal{D}}(GP) \rangle$, where GP is a graph pattern and $\mathcal{D}$ is a dataset. The main challenge is that there can be *too many graph patterns with useless 0 correspondence*. To avoid generating such meaningless query pairs, we make sure that the shared variables of two queries (or graph patterns) belong to the same type. Hence, we perform data summarisation based on the types of nodes in the graph. Given an RDF graph, the summarisation is to generate a condensed description which reduces the search space of the graph

pattern mining. Roughly speaking, the summarisation is an analogue to the schema, e.g. E-R diagram, in relational database system. This summarisation takes the form of graph patterns which reveal the possible relations among individuals. One of its good properties is that any query which is interesting by our definitions is a subgraph(or subgraphs) of the summarisation. This property allows us jump out from the "swamp" of original data graphs and focus on the summarisations to mine interesting queries. Furthermore, the size of the summarisation graph is extremely small by comparing to the original graph. The biggest summarisation graph in our test datasets only has 44 triples. Such tiny sized summarisations can largely facilitate the mining process e.g. expensive mining algorithms are applicable.

The second step computes the correspondences between graph patterns. The input of this step is the data summaries delivered by the previous step and the datasets. The output of this step is a set of 4-tuples $\langle GP_1, GP_2, support, confidence\rangle$ where GP_1 and GP_2 are graph patterns, $support$ and $confidence$ are used to characterise the correspondences between GP_1 and GP_2. Assuming GP_1 and GP_2 share a vector of variables v, then $support = |C_v(GP_1, GP_2)|$, $confidence = \frac{|C_v(GP_1,GP_2)|}{|I_v(GP_1)|}$ when $I_v(GP_1) > 0$ and 1 when $I_v(GP_1) = 0$. *support* indicates how frequent do GP_1 and GP_2 share instances w.r.t. v. The higher $support$ is, the more frequent the two graph patterns share instances on v in general. *confidence* indicates how frequent do instances of GP_1 w.r.t. v are also instances of GP_2. The higher $confidence(GP_1, GP_2)$ is, the more frequent that instances of GP_1 can be shared with GP_2. The challenge in this step is to *identify different patterns with desired correspondence*. In the domain of ontology learning, algorithms of inductive logic programming (ILP), association rule mining have been explored to find relationships between concepts and relations [Völker and Niepert, 2011, Lehmann et al., 2011]. Inspired by these works, we examine three different approaches (worst case polynomial time) to find the corresponding graph patterns. **FOIL** (First Order Inductive Learning) constructs the graph pattern by including a set of possible reachable variables via graduately growing the graph pattern. The algorithm selects a best variable from the set by a gain function (c.f. [Quinlan and Cameron-Jones, 1993]). FOIL tends to generate star-shaped graph patterns. **COMB** and **LOOP** approaches utilise the association rule mining technology. They tend to generate chain-shaped or looped graph patterns.

This third step generates insightful queries based on our discussion in Sec. 4.1. The input of this step is the datasets, and the computed correspondences between graph patterns. The output of this step will be a set of insightful queries or query pairs.

5 Query Answering Implementation

5.1 Entity Description Block

Entity-centric is a common view shared by many data modeling methods in application development. Inspired by this, we need to identify the entity-centric data units to be the basic units for data reuse.

From the entity-centric view, a basic unit should be a concise data unit describing an entity in the data source. Suppose e is the entity in question. Intuitively, a candidate data unit about e can be those triples in each of which e is the subject. However such a data unit is not a self-contained data unit due to the notion of blank node in RDF data model. Let us see an example illustrated in Fig. 3. For the entity *Tim*, there are two triples: $< Tim, assistant, Amy >$ and $< Tim, work, \_ : b02 >$. The blank node of _:b02 in the second triple introduces imperfect information. Evidently, such a data block is not appropriate for data reuse.

Based on above observation, the entity-centric unit needs to be concise and self-contained especially when blank nodes are involved. To achieve this, we adopte a notion of Entity Description Block (EDB).

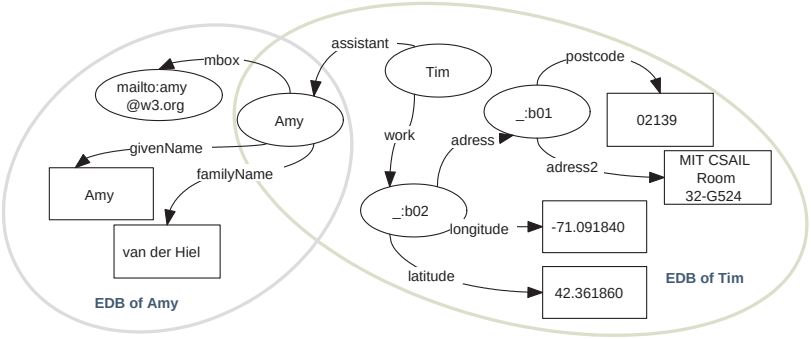

Figure 3: A fragment of Tim Berners-Lee's FOAF file

Definition 10 (ENTITY DESCRIPTION BLOCK (EDB)) *Let G be an RDF graph, we define $Sub(G) = \{s|\exists < s, p, o >\in G\}$. $\forall e \in U$, $EDB(G, e)$ is defined as follows.*

1. *If $e \notin Sub(G), EDB(G, e) = \emptyset$;*
2. *Otherwise, $EDB(G, e)$ is the minimal set of triples satisfying following conditions:*
   (a) *If $< e, p, o >\in G$, then $< e, p, o >\in EDB(G, e)$.*
   (b) *If $< e, p, o >\in G$, and $o \in B$, then $EDB(G, o) \subseteq EDB(G, e)$*

Given an RDF data source D, we use $G(D)$ to denote the set of all RDF triples contained in D. An RDF data source is viewed as a container of EDBs in definition 11.

Definition 11 (DATA SOURCE AS EDB CONTAINER) *Let U be the universe of URIs, D be viewed as a container of EDBs:*

$$EDB(D) = \{EDB(G(D), s)|s \in U, \exists < s, p, o >\in D\} \quad (1)$$

We design a naming scheme for identifying EDBs. Given an RDF data source, each entity description block can be assigned a global unique ID, which is composed of the data source URI and the entity URI. Evidently, this naming scheme gives the ability to extract EDBs from the Data Web by their IDs. In other words, the Web of Data can be viewed as a large set of EDBs, each of which can be located independently. Based on this, in the LOOD system, we implemented an EDB extraction service for users to download desired data portions. Although this service is now supported by a local cache for efficiency consideration, it can work well too without any local data support. All EDBs can be extracted from the Web of Data on the fly.

5.2 A Query Interface Based on Entity Description Patterns

We introduce a query interface for users to specify entity-centric data requirement. Users can specify constraints on text descriptions, entity types and properties. The core concept in this query interface is

called entity description pattern (EDP). This section first give the definition of an atomic EDP query and then introduce complex EDP queries which are disjunction expressions on atomic EDPs.

5.2.1 Atomic EDP Query

Let U be the universe of URIs, L be the universe of literals, an atomic EDP is in the format of (W, T, P), in which $W \subseteq L$, $T \subseteq U$, and $P \subseteq U$:

1. W - A string of words describing entities.
2. T - A set of URIs denoting entity types.
3. P - A set of URIs denoting properties used to describing the entities.

An EDP (an atomic EDP query if not specified especially) is to be matched on EDBs which are RDF graphs in essence. The semantics of an EDP query is twofold. In the first part, W is viewed as a keyword query. An EDB can also have a text description as $VDoc(EDB)$ (discussed in next section). Then the matching in this part is denoted as $Match(W, VDoc(EDB))$, which is evaluated in the same way of ordinary keyword-based query systems. In the second part, the semantics is composed of the constraints of entity types T and properties P. We denote it as $Match((T, P), EDB(G(D), e))$, which is evaluated to be true if and only if all the following conditions are satisfied.

1. $\forall t \in T, \exists < e, \textit{rdf:type}, t > \in EDB(G(D), e)$;
2. $\forall p \in P, \exists o_p, < e, p, o_p > \in EDB(G(D), e)$.

Finally, the matching between EDB and EDP is given in definition 12.

Definition 12 (MATCHING EDP AND EDB) *Given an EDP $\hat{p} = (W, T, P)$, an EDB $\hat{d} = EDB(G(D), e)$, the matching between them is evaluated as*

$$Match(\hat{p}, \hat{d}) = Match(W, VDoc(\hat{d})) \wedge Match((T, P), \hat{d}) \tag{2}$$

From the viewpoint of RDF data querying, the (T, P) constrains of an EDP query can also be viewed as an graph pattern (definition 13).

Definition 13 (GRAPH PATTERN VIEW OF EDP) *Given an EDP $\hat{p} = \{W, T, P\}$, let v_e be a variable, $\forall p \in P$, define a variable v_p, each v_p and v_e are distinct from each other. The graph pattern of $\hat{p}$ is,*

$$GP(\hat{p}) = \{< v_e, rdf : type, t > | t \in T >\} \cup \{< v_e, p, v_p > | p \in P\} \tag{3}$$

v_e is the variable to be bound.

As we know, a SPARQL query is also interpreted as a graph pattern [**?**]. Hence an EDP query with empty text description can be viewed as a SPARQL query. For example, the EDP query in Fig. **??** is represented in equation 4.

$$(\emptyset, \{DBpCls : City\}, \{DBpPrp : latitude, DBpPrp : longitude\}) \tag{4}$$

Fig. 4 depicts a SPARQL query which can be interpreted as the same graph pattern of the EDP query. The difference is in how the variable x is bound. In SPARQL, the variable is bound to a set of resources in one or a set of RDF data sets. In our scenario, it is bound to the IDs of EDBs in the portion of Data Web indexed in our system.

```
PREFIX DBpPrp:<http://dbpedia.org/property/>.
PREFIX DBpCls:<http://dbpedia.org/class/yago/>.
PREFIX rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?x
where {
        ?x rdf:type DBpCls:City.
        ?x DBpPrp:latitude ?y.
        ?x DBpPrp:longitude ?z.
      }
```

Figure 4: A SPARQL query as the same graph pattern of EDP query in (4).

5.3 Complex EDP Query

The query interface also provides users the ability to specify complex requirement based on EDPs. For example, we need to find a foaf:Person with a name information, in which either foaf:name or foaf:firstName is acceptable. We call this kind of query as complex EDPs. Before the definition of complex EDP, we define a boolean expression of URIs in EBNF syntax[**?**] (see definition 14). The OR operator represents logical disjunction. Given the definition of boolean expression on URIs, a complex EDP query is in the format of $(W, \{LE\}, \{LE\})$. Before query-answering, a complex EDP query will be transformed as a disjunction of atomic EDPs. In above example, the query is in the form of ($\emptyset$, {foaf:Person}, {'foaf:name OR foaf:firstName'}). This query is transformed into: ($\emptyset$, {foaf:Person}, {foaf:name}) OR ($\emptyset$, {foaf:Person}, {foaf:firstName}). Due to the limitation of space, details in the transformation will not be discussed.

Definition 14 *LE= URI, [{OR URI}];*

Evidently, with knowledge bases of synonyms corpus, ontologies and ontology mapping relationships, it is possible to automatically translate an application's information needs defined in DDL(data description language) into complex EDP queries. This automatic translation is an interesting and important work which is out of the scope of this paper.

5.4 EDP Query Answering

In our system, user requirements are expressed as EDP queries. The output is a set of data sources, each accompanied with the IDs of matched EDBs in it. Formally, the query-answering can be represented as 5.

$$Answering(\hat{p}) = \{(D, \{\hat{d} \in EDB(D) | Match(\hat{d}, \hat{p})\})\} \tag{5}$$

Essentially, both EDB and EDP are graph-structured. To implement the $Matcch$ function, a straightforward solution is to adopt graph-matching based query answering. However, such a solution is doubtful to be efficient enough to the web-scale.

We propose an structure-encoding based method to implement the $Match$ function. First the EDB and the EDP are encoded as term vectors. Then the similarity of two vectors are computed as the result. The key technique in this solution is the structure-encoding function defined in Definition 15. With the structure-encoding function we can encode the graph structure of either an EDB or an EDP into a vector.

Definition 15 (STRUCTURE-ENCODING FUNCTION) *Given a graph* $G \in (U \cup V \cup B) \times U \times (U \cup V \cup B \cup L)$, $\forall \tilde{s} \in U \cup V$

$$Encode(G, \tilde{s}) = \sum_{<\tilde{s}, rdf:type, t> \in G} (Vect(coinage('t', t))) + \sum_{<\tilde{s}, p, o> \in G, p \neq rdf:type} (Vect(coinage('p', p))) \quad (6)$$

$coinage(role, u)$ *is a function for creating a new word. The first parameter is a string denoting the role. The second is an URI. This function create a new word by concatenating the first string with the string format of the URI.* $Vect$ *is function to convert a string of words to a term vector.*

In addition to the graph pattern, an EDP also contains a text description, namely the W in (W, T, P). For matching such text description, we also need a text representation for an EDB. Virtual document based solutions were extensively adopted for such a problem[**?**, **?**, **?**]. Due to the limitation of space, we will not give the technique detail of constructing virtual documents for an RDF graph. We denote such a solution by a function of $VDoc(G)$. The input is an RDF graph and the output is a term vector.

So far, we can give the term vector representations of an EDP and an EDB in equation 7-8. Eventually, the $Match$ function in equation 5 is defined as Sim($\overrightarrow{\hat{p}}$, $\overrightarrow{\hat{d}}$), which is the similarity of vectors and follows the semantics of EDP query defined previously. To rank the result by data sources, the similarity is defined in equation 9.

$$\overrightarrow{\hat{p}} = Encode(GP(\hat{p})) + Vect(W) \quad (7)$$

$$\overrightarrow{\hat{d}} = Encode(\hat{d}) + VDoc(\hat{d}) \quad (8)$$

$$Sim(\{\hat{d}\}, \hat{p}) = \frac{\sum_{\hat{d}_i \in \{\hat{d}\}} (Sim(\overrightarrow{\hat{p}}, \overrightarrow{\hat{d}_i}))}{|\{\hat{d}\}|} \quad (9)$$

5.5 EDP Indexing

As defined in equation 5, the EDBs in the result is grouped by data sources. To achieve this, one possible strategy is to index EDBs separately and then group them at runtime. However, to group EDBs, the information of each EDB's container needs to be read. This requires intensive read operations on the secondary storage, which is too time consuming. We adopt a strategy of indexing data sources and propose a way to do querying and grouping at the same time.

As mentioned earlier, a data source is a set of EDBs and an EDB is modeled as a term vector. Therefore, a data source is modeled as a set of term vectors i.e. a matrix. This model introduces a big obstacle in indexing data sources. A logical-position based solution is introduced to address this issue. An inverted index can be viewed as the form of $term \longrightarrow \{(docid, weight)\}$. This index structure enables efficient similarity computation on term vectors. In such an index structure, if we can encode the structure information of a data source, it will be possible to utilize the efficiency of inverted index, without the needs to do additional read operations. Inspired by this, we assign a unique logical position to each EDB and then the data source can be viewed as an *extended term vector*. In such a vector, the entry in each dimension is turned from a number to a 2-tuple as (weight, a set of logical positions). Accordingly, the inverted index is extended in the form of $term \longrightarrow \{(docid, weight, logicalpositions)\}$.

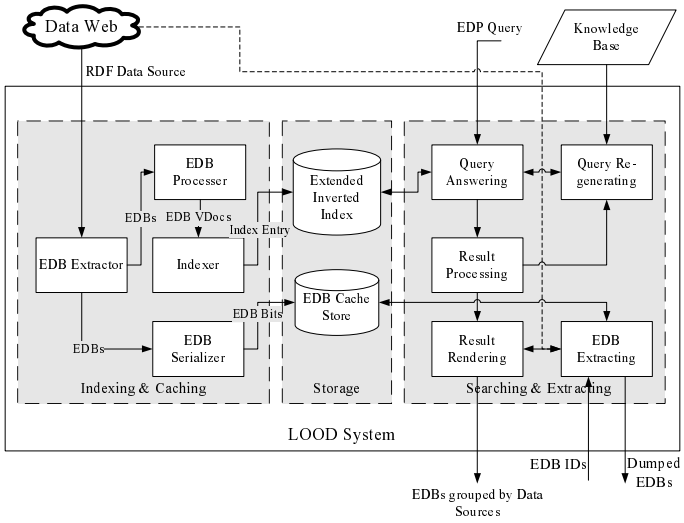

Figure 5: System Processes

The query answering is then turned to be two phases: index lookup and then disjunction on logical positions. Because all information needed is resulted from one read action, this indexing method enables performance potential to be satisfactory.

5.6 Architecture of Query Answering Engine

Here we describe the LOOD service. Fig. **??** shows the user interface. The input of this system is an EDP query. For building such a query, as shown in the upper part of the figure, an EDP query builder is designed. By this query builder, data consumers can specify three kinds of constraints on entity, namely text descriptions, types and properties. The output of the system is a set of matched EDBs grouped by data sources. The first matched EDB is depicted in the lower part of Fig. **??**. Each result consists of three parts. The first part is the data source URI. The second one is a detailed EDB which is most relevant to the query in the data source. As shown in the figure, each EDB is a data unit describing one specific entity. The property-value pairs are displayed and those matched pairs are highlighted. There is an **Export** button in the top right corner of the detailed EDB. Clicking it will fetch the EDB as an RDF document. The third part of a result is shown in the bottom of Fig. **??**. It tells how many EDBs are matched in this data source. There is also a **Export** button which is for fetching all matched EDBs in the data source.

5.7 Indexing and Caching

The indexing process is illustrated in the left part of Fig. 5. The *EDB Extractor* extracts all EDBs from a data source and sends them to the *EDB Processer*. The latter translates the EDB structure into a virtual document (VDoc). The resulted EDB VDocs are then sent to the *indxer*, which assigns a logical address to each VDoc and then encapsulates them into an index entry. All index entries are stored together as the *Extended Inverted Index.*

As shown in the lower left part of Fig. 5, an EDB caching process is also conducted in the meantime of indexing process. The *EDB Serializer* is the main component in such a process. Given EDBs as input, it serializes them into the *EDB Cache Store*. An EDB is the atomic unit in our system. Therefore, it is viewed simply as a sequence of bits and stored in a compressed manner. This makes it possible for our EDB cache store to be scalable to the scale of the Semantic Web.

5.8 Searching and Extracting

The searching process is shown in the right part of Fig. 5. Given an EDP query, the process proceeds as follows.

- **Query Preprocessing.** Given an EDP query, this process returns a set of data sources, each accompanied with IDs of matched EDBs within it. The queries specified by the query interface are often complex EDP queries (see section 3). This kind of queries can not be answered directly. Such a query needs to be preprocessed. The result of preprocessing is a disjunction of EDP queries (see section 3). After the preprocessing, the query expression is sent to the *Extended Inverted Index* and a set of data sources is returned. Accompanied with each data source, the IDs of matched EDBs are also identified. The results in the form of EDB IDs grouped by data sources are then sent to the *Result Processing* process.

- **Result Processing.** The input of this process is all matched results. The output is the top-K results. The number of EDBs is first checked. If it is less than a minimum threshold, a *Query Re-generating* process will be triggered. On the other hand, the result will be ranked according the formula (9). Then the top-K result will be sent to the *Result Rendering* process.

- **Query Re-generating.** This process is to do query expansion when the original query matches too few result. Detailed query expansion method will not be given in this paper. The re-generated query will be sent to the *Query Answering* process to start a new search.

Depicted in the bottom right part of Fig. 5, the *EDB Extracting* process is used for fetching EDBs as an RDF document. As mentioned above, this process can be called to render most relevant EDBs in the result page. In addition, it also serves as the EDB Export services which can be triggered by the user in the UI. The input of the process is a set of EDB IDs. The output is an RDF graph composed of corresponding EDBs. For each EDB ID, it will first search the *EDB Cache Store*. If local cache exists, it will be deserialized. If local cache is not available, it will search for it in the Data Web. The web-scale lookup is viable because EDB ID is web-accessible (see section 2). If the data source is an RDF document, it will first download the document and then extract the EDB according to the entity local ID. If the data source is a SPARQL endpoint, a SPARQL query will be submitted.

6 Evaluation

In this deliverable, we have two main tasks. One is query generation and the other is query answering. Accordingly, in this section, we present the evaluation results of query generation and query answering.

6.1 Query Generation Evaluation

We implemented the framework in Sec. 4.4 and evaluate its performance with benchmark datasets: **LUBM** (Lehigh University Benchmark) is an artificial dataset in which data is automatically generated. Its transparency makes it easier for us to examine whether the queries genreated by our system is useful or not. We generated 15,247 triples in our evaluation. **DBLP** is a large and real world dataset which includes bibliography data. In our evaluation, we used DBLP2011 data [4] (3,584,734 triples). **DBTune** hosts a selection of music-related RDF datasets. In our evaluation, we used the Jamedon [5] (1,047,950 triples) and BBC-PEEL [6] (271,369 triples) datasets. They are both using the music ontology [7] and the FOAF ontology [8] as terminologies.

6.1.1 Efficiency of EDP Graph

As described in previous sections, the EDP graph is the main output from the summarisation module. Its size determines the search space of the pattern mining stage so it is important to keep it minimal. In this subsection, we first introduce a formula to calculate the compression ratio of EDP graph over the original RDF graph. Later, we show the actual compression ratio on different datasets.

As mentioned in Sec. **??**, both the original datasets and our summarisations (as graph patterns) are graphs essentially and can be treated as sets of triples. We use $|G|$, the number of type and relation triples, to represent the size of a graph G. The compression ratio of EDP graph can then be defined as $r = \frac{|\mathcal{G}_{EDP(G)}|}{|G|}$. According to Def. **??**, each triple $t_{ij} =< v_i, l, v_j >$ in $\mathcal{G}_{EDP(G)}$ is mapped to a set of triples in the G: $\{< e_i, l, e_j > | v_i = rep(e_i), v_j = rep(e_j)\}$. We call the later as the instance of t_{ij}, denoted as $I_{t_{ij}}$. Then, the compression ratio $r = \frac{|\mathcal{G}_{EDP(G)}|}{\sum_{t_{ij} \in \mathcal{G}_{EDP(G)}} |I_{t_{ij}}|}$. From this formula, one can figure out that the size of each instance $I_{t_{ij}}$ is an important factor to the compression ratio. The average size of the instances is an indicator of the unity in one dataset. This value, denoted as u, means the average size of unity data blocks in one dataset. And its value is the reciprocal of the compression ratio.

Table 1: Dataset Statistics and Compression Ratio

Dataset		$\|G\|$	$\|\mathcal{G}\|$	r	u
DBLP2011		3,584,734	28	7.81E-06	128,026
LUBM		15,247	44	0.0028	193
DBTune	BBC-Peel	271,369	18	6.63E-05	15,076
	Jamendo	1,047,950	18	1.72E-05	58,219

In the second and third columns, Table 6.1.1 gives the statistics of the sizes (# triples) of original dataset and its EDP graph correspondingly. The compression ratios are list in the fourth column. It is notable that the values are quite small in all cases, indicating that the EDP graph are much smaller than its original data. One interesting observation is that the EDP graph of LUBM is larger than the one of DBLP2011, while originally DBLP2011 is much larger than LUBM. This implies that larger datasets do not necessarily have larger EDP graphs. This could be explained by the essence of the EDP graph which corresponds to the schema level information. This point can be further proved by the compression

[4] http://law.di.unimi.it/webdata/dblp-2011/
[5] http://dbtune.org/jamendo/
[6] http://dbtune.org/bbc/peel/
[7] http://musicontology.com/
[8] http://www.foaf-project.org/

ratios of the last two datasets (Jamendo and BBC-Peel). While the sizes of the two datasets are quite different, they have similar sized EDP graphs because they are in the same (music) domain. The last column shows the average size of unity data blocks of each dataset.

6.1.2 Query Generation Results

We applied our framework on the above datasets and generate queries on graph patterns with strongest support and confidence. Such generation can be performed efficiently. We examined the top 20 generated queries (query pairs) for each dataset and results showed that they are all meaningful. Some examples of generated queries from the LUBM dataset are presented in Table 2.

Table 2: Query examples.

	Query correspondences

1	`SELECT ?x`
	`WHERE {?x rdf:type lubm:ResearchGroup.}`
	
	`SELECT ?x`
	`WHERE {?x lubm:subOrganizationOf ?y.`
	`?y rdf:type lubm:Department.}`

	Query exceptions

2	`SELECT ?x`
	`WHERE {?x lubm:headOf ?y.`
	`?y rdf:type lubm:Department.`
	`FILTER NOT EXISTS`
	`{?x rdf:type lubm:FullProfessor}}`

	Looped queries

3	`SELECT ?x ?y ?z ?o`
	`WHERE {?x lubm:worksFor ?z.`
	`?x lubm:teacherOf ?y.`
	`?o lubm:memberOf ?z.`
	`?o lubm:takesCourse ?y.`
	`?o lubm:advisor ?x.}`

For example, query 1 suggests a high correspondence between the sub-organizations of some department and research groups. This reveals that a sub-organization of a department is very likely to be a research group. Such insights will be helpful when investigating the administration structures in universities. Query 2 investigates if the head of a department must be a full professors. In the evaluated dataset this indeed is the case. By automatically generating a query on cases where head of department is not known to be a full professor, users can easily identify potential exceptions. Another category of insightful queries are queries with loops, such as Query 3 in Table 2. This query suggests very high support and confidence that an advisor and an advisee work for (is a member of) the same entity, and the advisor is the teacher of some course that is taken by the advisee. This query contains three loops, and is very difficult to be expressed with normal ontology language.

In this section we evalaute different mining methods to discover graph patterns. Table 3 shows the comparison of them. All three methods were implemented based on the EDP summarisation and query answeirng. In COMB and LOOP the support threshold is 0. The results of LOOP are length 3. The time

Table 3: Comparison of FOIL, COMB and LOOP in terms of the # of triples, # of identified patterns, % of variable loop apperance, and the running time (-: out of time in 15 minutes) on LUBM / DBLP2011 / BBC-Peel / Jamendo.

method	average # of triples	# of identified patterns	% of variable loop appearnce	running time (in millisec)
FOIL	5.022 / - / 4.3 / 3.5	45 / - / 20 / 12	60% / - / 73.3% / 81.3%	15,408 / - / 50,821 / 134,530
COMB	4.87 / 4.965 / 3.559 / 2.875	215 / 142 / 59 / 40	100% for all	28,573 / 757,026 / 34,099 / 19,656
LOOP	3 for all	59 / 39 / 0 / 0	100% for all	4,186 / 793,909 / 1,231 / 3,746

Table 4: Indexing Time and Index Sizes on Three Data Sets

	DS1		DS2		DS3	
	time	size	time	size	time	size
LOOD	12.4 Min.	376 M	19 Min.	237 M	25.8 Min.	518 M
Sesame	4.5 Hours	454 M	6 Hours	758 M	7.5 Hours	970 M
Jena TDB	5.4 Hours	775 M	7.4 Hours	1.1 G	8.4 Hours	1.6 G

is collected by averaging 3 runs.

From Table 3 we can see that these methods are generally complementary. None of them is more efficient on all datasets. Generally, FOIL generates less but larger graph patterns and costs more time (had a time out in DBLP2011 due to its size). COMB and LOOP can both generate patterns with loops while LOOP is relatively less effective that it generates less such patterns on all datasets.

6.2 Query-answering Evaluation

To evaluate the performance of our indexing method, we compare it with two leading RDF database systems, namely Jena TDB and Sesame 2. The evaluation is conducted on three real world data sets, which are extracted from *Falcons 2008 data set* according to strategies described as follows.

- DS1: Including all RDF documents hosted in *dblp.l3s.de*. (#Docs:150K, #Triples:5.2M)
- DS2: Randomly selecting 200K documents hosted in *dbpedia.org*. (#Docs:200K, #Triples:5.1M)
- DS3: From each of the top-10 hosts (ranked by document numbers), randomly selecting 20K documents. (#Docs:200k, #Triples:10M)

For each data set, we designed two classes of queries, namely atomic EDP queries and complex EDP queries. Each class contains six queries, which were various in both constraint types and constraint numbers. As mentioned earlier, each EDP query can be viewed as a graph pattern. According to its graph pattern, for each EDP query, a SPARQL query was constructed, which was to be sent to Sesame 2 and Jena TDB for evaluation. We executed each query multiple times, eliminated outliers, and averaged the results. All the experiments were executed on a machine with a 2.6GHz Intel Quad CPU and 3GB of RAM. The indexing time and index sizes are shown in table 4. The query answering performance is illustrated in Fig. 6, in which the y axis is in log scale. As shown, our indexing method are both superior in indexing efficiency and query-answering performance. Especially, when the data set includes RDF data from various hosts (DS3), a significant performance improvement is observed. It is notable that our approach only works in answering EDP queries. So this evaluation is not to argue that our index

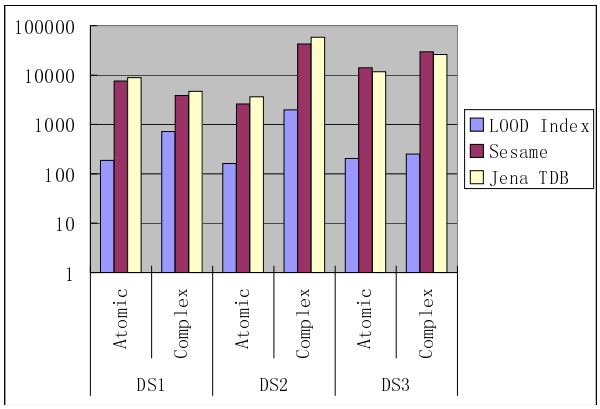

Figure 6: Query-answering Evaluation Result

method is superior than Sesame or Jena. Instead, the result confirms that our approach is efficient for EDP queries.

7 Conclusion

This deliverable investigates the problems of query generation and query answering in K-Drive project. To address these problems, the key enabling components in our approach are the EDP (entity description pattern) graph which represents the RDF dataset in a concise way, and the EDP index which supports very efficient EDP query operators.

For query generation, we follow the definition of insightful queries in Deliverable 3.1 a.k.a corresponding relations between pairs of graph patterns can be used to reveal interesting knowledge in datasets. Based on this definition, we introduced various association rule mining algorithms for searching such interesting knowledge. The EDP graph significantly reduces the search spaces for the association rule mining algorithm. In addition, the EDP index supports efficient rule mining operations. Our approach works for both pure query answering in RDF simple interpretations or reasoning based query answering (in e.g. OWL interpretation). For the latter case, materialisation needs to be performed before the construction of EDP. Our evaluation shows that the proposed summarisation approach is efficient and effective, and that the proposed framework can generate insightful queries from both synthetic and real world datasets.

The efficiency of the EDP index is achieved in twofolds. On the one hand the decomposition of conjunctive queries into atomic EDP operations makes it possible to implement the aromatic operations in an information retrieval manner. We proposed an extension structure from the traditional inverted index, which supports atomic EDP queries. On the other hand, the join operations of EDP queries are also implemented in an operation of co-occurrence searching. Such searching is also implemented by utilising the advantage of extended inverted index structure. Experiments show that our index method takes much less time in index construction and also is much more efficient than the state of the art query answering engines especially for star-shaped and chain-shaped queries.

Acknowledgement

This research has been funded by the European Commission within the 7th Framework Programme/Maria Curie Industry-Academia Partnerships and Pathways schema/PEOPLE Work Programme 2011 project K-Drive number 286348 (cf. `http://www.kdrive-project.eu`).

References

[Baader et al., 2010] Baader, F., Bienvenu, M., Lutz, C., Wolter, F., et al. (2010). Query and predicate emptiness in description logics. *Proc. of KR2010*.

[Chatzopoulou et al., 2009] Chatzopoulou, G., Eirinaki, M., and Polyzotis, N. (2009). Query Recommendations for Interactive Database Exploration.

[d'Aquin and Motta, 2011] d'Aquin, M. and Motta, E. (2011). Extracting Relevant Questions to an RDF Dataset Using Formal Concept Analysis. In *Proceedings of the sixth international conference on Knowledge capture*, pages 121–128.

[Fokoue et al., 2012] Fokoue, A., Meneguzzi, F., Sensoy, M., and Pan, J. (2012). Querying linked ontological data through distributed summarization. In *AAAI2012*.

[Görlitz et al., 2012] Görlitz, O., Thimm, M., and Staab, S. (2012). Splodge: Systematic generation of sparql benchmark queries for linked open data. In *Proc. of The Semantic Web–ISWC 2012*, pages 116–132. Springer.

[Grau and Stoilos, 2011] Grau, B. C. and Stoilos, G. (2011). What to Ask to an Incomplete Semantic Web Reasoner? In *IJCAI*, pages 2226–2231.

[Guo et al., 2005] Guo, Y., Pan, Z., and Heflin, J. (2005). LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics*, 3(2):158–182.

[Heino and Pan.,] Heino, N. and Pan., J. Z. RDFS Reasoning on Massively Parallel Hardware. In *Proc. of ISWC2012*.

[Kollia et al., 2011] Kollia, I., Glimm, B., and Horrocks, I. (2011). Sparql query answering over owl ontologies. *The Semantic Web: Research and Applications*, pages 382–396.

[Lehmann et al., 2011] Lehmann, J., Auer, S., Bühmann, L., and Tramp, S. (2011). Class expression learning for ontology engineering. *Journal of Web Semantics*, 9:71–81.

[Mishra et al., 2008] Mishra, C., Koudas, N., and Zuzarte, C. (2008). Generating Targeted Queries for Database Testing. In *In Proc. of ACM SIGMOD*, pages 499–510.

[Motik et al., 2008] Motik, B., Grau, B., and Sattler, U. (2008). Structured objects in owl: Representation and reasoning. In *Proc. WWW*, pages 21–25. Citeseer.

[Quinlan and Cameron-Jones, 1993] Quinlan, J. and Cameron-Jones, R. (1993). Foil: A midterm report. In *Machine Learning: ECML-93*, pages 1–20. Springer.

[Slutz, 1998] Slutz, D. (1998). Massive Stochastic Testing of SQL. In *In VLDB*, pages 618–622.

[Völker and Niepert, 2011] Völker, J. and Niepert, M. (2011). Statistical schema induction. *The Semantic Web: Research and Applications*.

[Zhang and Nasraoui, 2006] Zhang, Z. and Nasraoui, O. (2006). Mining search engine query logs for query recommendation. In *Proceedings of the 15th international conference on World Wide Web*, pages 1039–1040.