SemLAV: Local-As-View Mediation for SPARQL Queries

Gabriela Montoya, Luis Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, Maria-Esther Vidal

SemLAV: Local-As-View Mediation for SPARQL Queries

Gabriela Montoya*[1], Luis-Daniel Ibáñez[1], Hala Skaf-Molli[1], and Pascal Molli[1]
Maria-Esther Vidal[2]

[1] LINA– Nantes University. Nantes, France
{gabriela.montoya,luis.ibanez,hala.skaf,
pascal.molli}@univ-nantes.fr
[2] Universidad Simón Bolívar. Caracas, Venezuela
mvidal@ldc.usb.ve

Abstract. The Local-As-View (LAV) integration approach aims at querying heterogeneous data in dynamic environments. In LAV, data sources are described as views over a global schema which is used to pose queries. Query processing requires to generate and execute query rewritings, but for SPARQL queries, the LAV query rewritings may not be generated or executed in a reasonable time.
In this paper, we present SemLAV, an alternative technique to process SPARQL queries over a LAV integration system without generating rewritings. SemLAV executes the query against a partial instance of the global schema which is built on-the-fly with data from the relevant views. The paper presents an experimental study for SemLAV, and compares its performance with traditional LAV-based query processing techniques. The results suggest that SemLAV scales up to SPARQL queries even over a large number of views, while it significantly outperforms traditional solutions.

Keywords: Semantic Web · Data Integration · Local-as-view · SPARQL Query

1 Introduction

Processing queries over a set of autonomous and semantically heterogeneous data sources is a challenging problem. Particularly, a great effort has been done by the Semantic Web community to integrate datasets into the Linked Open Data (LOD) cloud [1] and make these data accessible through SPARQL endpoints which can be queried by federated query engines. However, there are still a large number of data sources and Web APIs that are not part of the LOD cloud. As consequence, existing federated query engines cannot be used to integrate these data sources and Web APIs. Supporting SPARQL query processing over these environments would extend federated query engines into the deep Web.

Two main approaches exist for data integration: data warehousing and mediators. In data warehousing, data are transformed and loaded into a repository;

* unit UMR6241 of the Centre National de la Recherche Scientifique (CNRS)

this approach may suffer from freshness problem [2]. In the mediator approach, there is a global schema over which the queries are posed and views describe data sources. Three main paradigms are proposed: Global-As-View (GAV), Local-As-View (LAV) and Global-Local-As-View (GLAV). In GAV mediators, entities of the global schema are described using views over the data sources, including or updating data sources may require the modification of a large number of views [3]. Whereas, in LAV mediators, the sources are described as views over the global schema, adding new data sources can be easily done [3]. Finally, GLAV is a hybrid approach that combines both LAV and GAV approaches. GAV is appropriate for query processing in stable environments. A LAV mediator relies on a query rewriter to translate a mediator query into the union of queries against the views. Therefore, it is more suitable for environments where data sources frequently change. Despite of its expressiveness and flexibility, LAV suffers from well-known drawbacks: (i) existing LAV query rewriters only manage conjunctive queries, (ii) the query rewriting problem is NP-complete for conjunctive queries, and (iii) the number of rewritings may be exponential.

SPARQL queries exacerbate LAV limitations, even in presence of conjunctions of triple patterns. For example, in a traditional database system, a LAV mediator with 140 conjunctive views can generate 10,000 rewritings for a conjunctive query with eight subgoals [4]. In contrast, the number of rewritings for a SPARQL query can be much larger. SPARQL queries are commonly comprised of a large number of triple patterns and some may be bound to *general predicates* of the RDFS or OWL vocabularies, e.g., `rdf:type`, `owl:sameAs` or `rdfs:label`, which are usually used in the majority of the data sources. Additionally, queries can be comprised of several star-shaped sub-queries [5]. Finally, a large number of variables can be projected out. All these properties impact the complexity of the query rewriting problem, even enumerating query rewritings can be unfeasible. For example, a SPARQL query with 12 triple patterns that comprises three star-shaped sub-queries can be rewritten using 476 views in billions of rewritings. This problem is even more challenging considering that statistics may be unavailable, and there are no clear criteria to rank or prune the generated rewritings [6]. It is important to note that for conjunctive queries, GLAV query processing tasks are at least as complex as LAV tasks [7].

In this paper, we focus on the LAV approach, and propose SemLAV, the first scalable LAV-based approach for SPARQL query processing. Given a SPARQL query Q on a set M of LAV views, SemLAV selects relevant views for Q and ranks them in order to maximize query results. Next, data collected from selected views are included into a partial instance of the global schema, where Q can be executed whenever new data is included; and thus, SemLAV incrementally produces query answers. Compared to a traditional LAV approach, SemLAV avoids generating rewritings which is the main cause of the combinatorial explosion in traditional rewriting-based approaches; SemLAV also supports the execution of SPARQL queries. The performance of SemLAV is no more dependent on the number of rewritings, but it does depend on the number and size of relevant views. Space required to temporarily include relevant views in the global schema instance

may be considerably larger than the space required to execute all the query rewritings one by one. Nevertheless, executing the query once on the partial instance of the global schema could produce the answers obtained by executing all the rewritings.

To empirically evaluate the properties of SemLAV, we conducted an experimental study using the Berlin Benchmark [8] and queries and views designed by Castillo-Espinola [9]. Results suggest that SemLAV outperforms traditional LAV-based approaches with respect to answers produced per time unit, and provides a scalable LAV-based solution to the problem of executing SPARQL queries over heterogeneous and autonomous data sources.

The contributions of this paper are the following:

- Formalization of the problem of finding the set of relevant LAV views that maximize query results; we call this problem MaxCov.
- A solution to the MaxCov problem.
- A scalable and effective LAV-based query processing engine to execute SPARQL queries, and to produce answers incrementally.

The paper is organized as follows: Section 2 presents basic concepts, definitions and a motivating example. Section 3 defines the MaxCov problem, SemLAV query execution approach and algorithms. Section 4 reports our experimental study. Section 5 summarizes related work. Finally, conclusions and future work are outlined in Section 6.

2 Preliminaries

Mediators are components of the mediator-wrapper architecture [10]. They provide an uniform interface to autonomous and heterogeneous data sources. Mediators also rewrite an input query into queries against the data sources, and merge data collected from the selected sources. Wrappers are software components that solve interoperability between sources and mediators by translating data collected from the sources into the schema and format understood by the mediators; the schema exposed by the wrappers is part of the schema exposed by its corresponding mediator.

The problem of processing a query Q over a set of heterogeneous data sources corresponds to answer Q using the instances of these sources. Although this problem has been extensively studied by the Database community [11], it has not been addressed for SPARQL queries. The following definitions are taken from Database existing solutions. Many of them are given for conjunctive queries. A conjunctive query has the form: $\mathrm{Q}(\bar{X})$:- $p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$, where p_i is a predicate, $\bar{X}_i$ is a list of variables and constants, $\mathrm{Q}(\bar{X})$ is the head of the query, $p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$ is the body of the query, and each element of the body is a query subgoal. In a conjunctive query, distinguished variables are variables that appear in the head, they should also appear in the body. Variables that appear in the body, but not in the head are existential variables.

Definition 1 (LAV Integration System [12]). *A LAV integration system is a triple IS=$< G, S, M >$ where G is a global schema, S is a set of sources or source schema, and M is a set of views that map sources in S into the global schema G.*

For the rest of the paper, we assume that views in M are limited to conjunctive queries. Both views and mediator queries are defined over predicates in G.

Definition 2 (Sound LAV View [12]). *Given IS=$< G, S, M >$ a LAV integration system, and a view v_i in M. The view v_i is sound if for all instance $I(v_i)$ of v_i, and all D virtual database instance of G, $I(v_i)$ is contained in the evaluation of view v_i over D, i.e., $I(v_i) \subseteq v_i(D)$.*

Definition 3 (Query Containment and Equivalence [13]). *Given two queries $Q1$ and $Q2$ with the same number of arguments in their heads, $Q1$ is contained in $Q2$, $Q1 \sqsubseteq Q2$, if for any database instance D the answer of $Q1$ over D is contained in the answer to $Q2$ over D, $Q1(D) \subseteq Q2(D)$. $Q1$ is equivalent to $Q2$ if $Q1 \sqsubseteq Q2$ and $Q2 \sqsubseteq Q1$.*

Definition 4 (Containment Mapping [13]). *Given two queries $Q1$ and $Q2$, $\bar{X}$ and $\bar{Y}$ the head variables of $Q1$ and $Q2$ respectively, and ψ a variable mapping from $Q1$ to $Q2$, ψ is a containment mapping if $\psi(\bar{X}) = \bar{Y}$ and for every query subgoal $g(\bar{X}_i)$ in the body of $Q1$, $\psi(g(\bar{X}_i))$ is a subgoal of $Q2$.*

Theorem 1 (Containment [13]). *Let $Q1$ and $Q2$ be two conjunctive queries, then there is a containment mapping from $Q1$ to $Q2$ if and only if $Q2 \sqsubseteq Q1$.*

Definition 5 (Query Unfolding [13]). *Given a query Q and a query subgoal $g_i(\bar{X}_i)$, $g_i(\bar{X}_i) \in body(Q)$, where g_i corresponds to a view: $g_i(\bar{Y})$:-$s_1(\bar{Y_1}), \ldots, s_n(\bar{Y_n})$, the unfolding of g_i in Q is done using a mapping τ from variables in $\bar{Y}$ to variables in $\bar{X}_i$, replacing $g_i(\bar{X}_i)$ by $s_1(\tau(\bar{Y_1})), \ldots, s_n(\tau(\bar{Y_n}))$ in Q. Variables that occur in the body of g_i but not in $\bar{X}_i$ are replaced by fresh (unused) variables by mapping τ.*

Definition 6 (Equivalent Rewriting [11]). *Let Q be a query and $M = \{v_1, \ldots, v_m\}$ be a set of views definitions. The query Q' is an equivalent rewriting of Q using M if:*

- *Q' refers only to views in M, and*
- *Q' is equivalent to Q.*

Definition 7 (Maximally-Contained Rewriting [11]). *Let Q be a query, $M = \{v_1, \ldots, v_m\}$ be a set of views definitions, and L be a query language[3]. The query Q' is a maximally-contained rewriting of Q using M with respect to L if:*

- *Q' is a query in L that refers only to the views in M,*

[3] L is a query language defined over the alphabet composed of the global and source schema

– Q' *is contained in* Q*, and*
– *there is no rewriting* $Q_1 \in L$*, such that* $Q' \sqsubseteq Q_1 \sqsubseteq Q$ *and* Q_1 *is not equivalent to* Q'*.*

Theorem 2 (Number of Candidate Rewritings [2]). *Let* N*,* O *and* M *be the number of query subgoals, the maximal number of views subgoals, and the set of views, respectively. The number of candidate rewritings in the worst case is:* $(O \times |M|)^N$*.*

Theorem 3 (Complexity of Finding Rewritings [11]). *The problem of finding an equivalent rewriting is NP-complete.*

Consider L in Definition 7 as the union of conjunctive queries, then view v would be used to answer query Q if there is one conjunctive query $r \in Q'$ such that v appears as the relation of one of r query subgoals. As $Q' \sqsubseteq Q$, then $r \sqsubseteq Q$. View v is called a relevant view atom. The next definition formalizes this notion.

Definition 8 (Relevant View Atom [13]). *A view atom* v *is relevant for a query atom* g *if one of its subgoals can play the role of* g *in the rewriting. To do that, several conditions must be satisfied: (1) the view subgoal should be over the same predicate as* g*, and (2) if* g *includes a distinguished variable of the query, then the corresponding variable in* v *must be a distinguished variable in the view definition.*

The concepts of relevant view and coverage have been widely used in the literature [13, 11]; nevertheless, they have been introduced in an informal way. The following definitions precise the properties that are assumed in this paper.

Definition 9 (Relevant Views). *Let* Q *be a conjunctive query,* $M = \{v_1, \ldots, v_m\}$ *be a set of view definitions, and* q *be a query subgoal, i.e.,* $q \in body(Q)$*. The set of relevant views for* q *corresponds to the set of relevant view atoms for the query subgoal* q*, i.e.,* $RV(M, q) = \{\tau(v) : v \in M \wedge w \in body(v) \wedge \psi(q) = \tau(w) \wedge (\forall x : x \in Vars(q) \wedge distinguished(x, Q) : distinguished(x, v))\}$[4]*. The set of relevant views for* Q *corresponds to the views that are relevant for at least one query subgoal, i.e.,* $RV(M, Q) = \{\tau(v) : q \in body(Q) \wedge v \in M \wedge w \in body(v) \wedge \psi(q) = \tau(w) \wedge (\forall x : x \in Vars(q) \wedge distinguished(x, Q) : distinguished(x, v))\}$*.*

Definition 10 (Coverage). *Let* Q *be a conjunctive query,* v *be a view definition,* q *be a query subgoal, and* w *be a view subgoal. The predicate* $covers(w, q)$ *holds if and only if* w *can play the role of* q *in a query rewriting.*

We illustrate some of the given definitions for the LAV-based query rewriting approach using SPARQL queries. This will provide evidence of the approach limitations even for simple queries. In the following example, the global schema G is defined over the Berlin Benchmark [8] vocabulary. Consider a SPARQL query Q on G; Q has seven subgoals and returns information about products as

[4] $\psi(q)$ corresponds to the application of ψ to the variables of q (idem for $\tau(w)$).

shown in Listing 1.1. Listing 1.3 presents Q as a conjunctive query, where triple patterns are represented as query subgoals.

```
SELECT *
WHERE {
  ?X1 rdfs:label ?X2 .
  ?X1 rdfs:comment ?X3 .
  ?X1 bsbm:productPropertyTextual1 ?X8 .
  ?X1 bsbm:productPropertyTextual2 ?X9 .
  ?X1 bsbm:productPropertyTextual3 ?X10 .
  ?X1 bsbm:productPropertyNumeric1 ?X11 .
  ?X1 bsbm:productPropertyNumeric2 ?X12 .
}
```

Listing 1.1: SPARQL query Q

```
SELECT *
WHERE {
  ?X1 rdfs:label ?X2 .
  ?X1 rdf:type ?X3 .
  ?X1 bsbm:productFeature ?X4
        .
}
```

Listing 1.2: SPARQL View $s1$

```
Q(X1, X2, X3, X8, X9, X10, X11, X12) :- label(X1, X2), comment(X1, X3),
     productPropertyTextual1(X1, X8), productPropertyTextual2(X1, X9),
     productPropertyTextual3(X1, X10), productPropertyNumeric1(X1, X11),
     productPropertyNumeric2(X1, X12)
```

Listing 1.3: Q expressed as a conjunctive query

```
s1(X1,X2,X3,X4):-label(X1,X2),type(X1,X3),productfeature(X1,X4)
s2(X1,X2,X3):-type(X1,X2),productfeature(X1,X3)
s3(X1,X2,X3,X4):-producer(X1,X2),label(X2,X3),publisher(X1,X2),
     productfeature(X1,X4)
s4(X1,X2,X3):-productfeature(X1,X2),label(X2,X3)
s5(X1,X2,X3,X4,X5,X6,X7):-label(X1,X2),comment(X1,X3),producer(X1,X4),
     label(X4,X5),publisher(X1,X4),productpropertytextual1(X1,X6),
     productpropertynumeric1(X1,X7)
s6(X1,X2,X3,X4,X5):-label(X1,X2),product(X3,X1),price(X3,X4),vendor(X3,X5)
s7(X1,X2,X3,X4,X5,X6):-label(X1,X2),reviewfor(X3,X1),reviewer(X3,X4),
     name(X4,X5),title(X3,X6)
s9(X1,X2,X3,X4):-reviewfor(X1,X2),title(X1,X3),text(X1,X4)
s10(X1,X2,X3):-reviewfor(X1,X2),rating1(X1,X3)
s11(X1,X2,X3,X4,X5,X6,X7):-label(X1,X2),comment(X1,X3),producer(X1,X4),
     label(X4,X5),publisher(X1,X4),productpropertytextual2(X1,X6),
     productpropertynumeric2(X1,X7)
s12(X1,X2,X3,X4,X5,X6,X7):-label(X1,X2),comment(X1,X3),producer(X1,X4),
     label(X4,X5),publisher(X1,X4),productpropertytextual3(X1,X6),
     productpropertynumeric3(X1,X7)
s13(X1,X2,X3,X4,X5,X6,X7):-label(X1,X2),product(X3,X1),price(X3,X4),
     vendor(X3,X5),offerwebpage(X3,X6),homepage(X5,X7)
s14(X1,X2,X3,X4,X5,X6,X7):-label(X1,X2),product(X3,X1),price(X3,X4),
     vendor(X3,X5),deliverydays(X3,X6),validto(X3,X7)
s15(X1,X2,X3,X4,X5,X6,X7,X8,X9):-product(X1,X2),price(X1,X3),vendor(X1,X4),
     label(X4,X5),country(X4,X6),publisher(X1,X4),reviewfor(X7,X2),
     reviewer(X7,X8),name(X8,X9)
```

Listing 1.4: Views s1-s10 from [9]

Consider M composed of 14 data sources defined as conjunctive views over the global schema G as in Listing 1.4; the Berlin Benchmark [8] vocabulary terms are represented as binary predicates in the conjunctive queries that define the data sources. Source *s1* can be defined as in Listing 1.2; note that we have done just a syntactic translation from this SPARQL query to the conjunctive query presented in Listing 1.4.

For instance, *s1* retrieves information about product type, label and product feature. The `rdfs:label` predicate is a *general predicate*. Commonly, general predicates are part of the definition of many data sources, and the number of rewritings of SPARQL queries that comprise triple patterns bound to general predicates can be very large. The general predicate `rdfs:label` in query Q can be mapped to views *s1, s3-s7, s11-s15*.

```
r(X1,X2,X3,X8,X9,X10,X11,X12) :- s6(X1,X2,_0,_1,_2),
    s5(X1,_3,X3,_4,_5,_6,_7), s5(X1,_8,_9,_10,_11,X8,_12),
    s11(X1,_13,_14,_15,_16,X9,_17), s12(X1,_18,_19,_20,_21,X10,_22),
    s5(X1,_23,_24,_25,_26,_27,X11), s11(X1,_28,_29,_30,_31,_32,X12)
```

Listing 1.5: A query rewriting for Q

Listing 1.5 presents a query rewriting for Q, its subgoals cover each of the query subgoals of Q, e.g., $s6(X1, X2, \_0, \_1, \_2)$ covers the first query subgoal of Q, $label(X1, X2)$. $\psi(label(X1, X2)) = \tau(label(X1, X2))$; the mapping τ from view variables to rewriting variables is: $\tau(X1) = X1$, $\tau(X2) = X2$, $\tau(X3) = \_0$, $\tau(X4) = \_1$, $\tau(X5) = \_2$, and the mapping ψ from query variables to rewriting variables is: $\psi(Xi) = Xi$, for all Xi in the query head. Then, view $s6(X1, X2, \_0, \_1, \_2)$ is relevant for answering the first query subgoal of Q. Notice that third, fourth and fifth projected variables of $s6$ correspond to existential variables because they are not relevant to cover the first query subgoal of Q with $s6$.

To illustrate how the number of rewritings for Q can be affected by the number of data sources that use the general predicate `rdfs:label`, we run the LAV query rewriter MCDSAT [14].[5] First, if 14 data sources are considered, Q can be rewritten in 42 rewritings. For 28 data sources, there are 5,376 rewritings, and 1.12743e+10 rewritings are generated for 224 sources.[6] With one simple query, we can illustrate that the number of rewritings can be extremely large, being in the worst case exponential in the number of query subgoals and views. In addition to the problem of enumerating this large number of query rewritings, the time needed to compute them may be excessively large. Even using reasonable timeouts, only a small number of rewritings may be produced.

Table 1 shows the number of rewritings obtained by the state-of-the-art LAV rewriters GQR[4], MCDSAT[14] and MiniCon[15], when 224 views are considered for Q and timeouts are set up to 5, 10 and 20 minutes. Note that all these rewriters are able to produce only empty results or a small number of rewritings.

In summary, even if the LAV approach constitutes a flexible approach to integrate data from heterogeneous data sources, query rewriting and processing tasks may be unfeasible in the context of SPARQL queries. Either the number of query rewritings is too large to be enumerated or executed in a reasonable time. To overcome these limitations and make feasible the LAV approach for SPARQL

[5] MCDSAT [14] is the only query rewriting tool publicly available that counts the number of rewritings without enumerating all of them.

[6] The 14 data sources setup is defined as in Listing 1.4, the one with 28 data sources has two views for each of the views in Listing 1.4, and the one with 224 sources has 16 views for each of the views in Listing 1.4

Table 1: Number of rewritings obtained from the rewriters GQR, MCDSAT and MiniCon with timeouts of 5, 10 and 20 minutes. Using 224 views and query Q

Rewriter	5 minutes	10 minutes	20 minutes
GQR	0	0	0
MCDSAT	211,125	440,308	898,766
MiniCon	0	0	0

queries, we propose a novel approach named SemLAV. SemLAV identifies and ranks the relevant views of a query, and executes the query over the data collected from the relevant views; thus, SemLAV is able to output a high proportion of the answer in a short time.

3 The SemLAV Approach

SemLAV is a scalable LAV-based approach for processing SPARQL queries. It is able to produce answers even for SPARQL queries against large integration systems with no statistics. SemLAV follows the traditional mediator-wrapper architecture [10]. Schemas exposed by the mediators and wrappers are expressed as RDF vocabularies. Given a SPARQL query Q over a global schema G and a set of sound views $M = \{v_1, \ldots, v_m\}$, SemLAV executes the original query Q rather than generating and executing rewritings as in traditional LAV approaches. SemLAV builds an instance of the global schema on-the-fly with data collected from the relevant views. The relevant views are considered in an order that enables to produce results as soon as the query Q is executed against this instance.

Contrary to traditional wrappers which populate structures that represent the heads of the corresponding views, SemLAV wrappers return RDF Graphs composed of the triples that match the triple patterns in the definition of the views. SemLAV wrappers could be more expensive in space than the traditional ones. However, they ensure that original queries are executable even for full SPARQL queries and they make query execution dependent on the number of views rather than on the number of rewritings.

To illustrate the SemLAV approach, consider a SPARQL query Q with four subgoals:

```
SELECT *
WHERE {
   ?Offer bsbm:vendor ?Vendor .
   ?Vendor rdfs:label ?Label .
   ?Offer bsbm:product ?Product .
   ?Product bsbm:productFeature ?ProductFeature .
}
```

and a set M of five views:

```
v1(P,L,T,F):-label(P,L),type(P,T),productfeature(P,F)
v2(P,R,L,B,F):-producer(P,R),label(R,L),publisher(P,B),productfeature(P,F)
v3(P,L,O,R,V):-label(P,L),product(O,P),price(O,R),vendor(O,V)
v4(P,O,R,V,L,U,H):-product(O,P),price(O,R),vendor(O,V),label(V,L),
   offerwebpage(O,U),homepage(V,H)
v5(O,V,L,C):-vendor(O,V),label(V,L),country(V,C)
```

In the traditional LAV approach, 60 rewritings are generated and the execution of all these rewritings will produce all possible answers. However, this is time-consuming and uses a non-negligible amount of memory to store data collected from views present in the rewritings. In case there are not enough resources to execute all these rewritings, as many rewritings as possible would be executed. We apply a similar idea in SemLAV, if it is not possible to consider the whole global schema instance to ensure a complete answer, then a partial instance will be built. The partial instance will include data collected from as many relevant views as the available resources allow.

The execution of the query over this partial schema instance will cover the results of executing a number of rewritings. The number of rewritings covered by the execution of Q over the partial schema instance could be exponential in the number of views included in the instance. Therefore, the size of the set of covered rewritings may be even greater than the number of rewritings executable in the same amount of time.

Table 2: Impact of the different views ordering on the number of covered rewritings

# Included views (k)	Order One		Order Two	
	Included views (V_k)	# Covered rewritings	Included views (V_k)	# Covered rewritings
1	v5	0	v4	0
2	v5, v1	0	v4, v2	2
3	v5, v1, v3	6	v4, v2, v3	12
4	v5, v1, v3, v2	8	v4, v2, v3, v1	32
5	v5, v1, v3, v2, v4	60	v4, v2, v3, v1, v5	60

The order in which views are included in the partial global schema instance impacts the number of covered rewritings. Consider two different orders for including the views of the above example: v5, v1, v3, v2, v4 and v4, v2, v3, v1, v5. Table 2 considers partial global schema instances of different sizes. For each partial global schema instance, the included views and the number of covered rewritings are presented. Executing Q over the growing instances corresponds to the execution of a quite different number of rewritings. For instance, if only four views could be included with the available resources, one order corresponds to the execution of 32 rewritings while the another one corresponds to the execution of only eight rewritings. If all relevant views for query Q could be included, then a

complete answer will be produced. However, the number of relevant views could be considerably large, therefore, if we only have resources to consider k relevant views, V_k, we should consider the ones that increase the chances of obtaining answers. With no knowledge about data distribution, we can only suppose that each rewriting has nearly the same chances of producing answers. Thus, the chances of obtaining answers are proportional to the number of rewritings covered by the execution of Q over an instance that includes views in V_k.

Maximal Coverage Problem (MaxCov). Given an integer $k > 0$, a query Q on a global schema G, a set M of sound views over G, and a set R of conjunctive queries whose union is a maximally-contained rewriting of Q in M. The Maximal Coverage Problem is to find a subset V_k of M comprised of k relevant views for Q, $V_k \subseteq M \wedge (\forall v : v \in V_k : v \in RV(Q, M)) \wedge |V_k| = k$, such that the set of rewritings covered by V_k, $Coverage(V_k, R)$, is maximal for all subsets of M of size k, i.e., there is no other set of k views that can cover more rewritings than V_k. $Coverage(V_k, R)$ is defined as:

$$Coverage(V_k, R) = \{r : r \in R \wedge (\forall p : p \in body(r) : p \in V_k)\} \quad (1)$$

The MaxCov problem has as an input a solution to the Maximally-Contained Rewriting problem. Nevertheless, using this for building a MaxCov solution would be unreasonable since it makes the MaxCov solution at least as expensive as the rewriting generation. Instead of generating the rewritings, we define a formula that estimates the number of covered rewritings when Q is executed over a global schema instance that includes a set of views. It is the product of the number of ways each query subgoal can be covered by the set of views. For a query $Q(\bar{X})$:- $p_1(\bar{X}_1), \ldots p_n(\bar{X}_n)$ using only views in V_k this formula is expressed as:

$$NumberOfCoveredRewritings(Q, V_k) = \Pi_{1 \leq i \leq n} |Use(V_k, p_i(\bar{X}_i))|, \quad (2)$$

where $Use(V_k, p) = \Sigma_{v \in V_k} \Sigma_{w \in body(v) \wedge covers(w,p)} 1$. This formula computes the exact number of covered rewritings when all the view variables are distinguished; this is because the coverage of each query subgoal by a given view can be considered in isolation. Otherwise, this expression corresponds to an upper bound of the number of covered rewritings of Q with respect to V_k.

Consider the second proposed ordering of the views in the above example, the numbers of views in V_4 that cover each query subgoal are:

- two for the first query subgoal (v4 and v3),
- four for the second query subgoal (v4, v2, v3 and v1),
- two for the third query subgoal (v4 and v3), and
- two for the fourth query subgoal (v2 and v1).

Thus, the number of covered rewritings is 32 ($2 \times 4 \times 2 \times 2$).

Next, we detail a solution to the MaxCov problem under the assumption that views only contain distinguished variables.

3.1 The SemLAV Relevant View Selection and Ranking Algorithm

The relevant view selection and ranking algorithm finds the views that cover each subgoal of a query. This algorithm creates a bucket for each query subgoal q, where a bucket is a set of relevant views; this resembles the first step of the Bucket algorithm[11]. Additionally, the algorithm sorts the buckets views according to the number of covered subgoals. Hence, the views that are more likely to contribute to the answer will be considered first. This algorithm is defined in Algorithm 1.

Algorithm 1 The Relevant View Selection and Ranking

```
Input: Q : SPARQL Query; M: Set of Views defined as conjunctive queries
Output: Buckets: Predicate → List<View>
  for all q ∈ body(Q) do
    buckets(q) ← ∅
  end for
  for all q ∈ body(Q) do
    b ← buckets(q)
    for all v ∈ M do
      for all w ∈ body(v) do
        if There are mappings τ, ψ, such that ψ(q) = τ(w) then
          v_i ← λ(v) {λ(v) replaces all variables a_i in the head of v by τ(a_i)}
          insert(b, vi) {add v_i to the bucket if it is not redundant}
        end if
      end for
    end for
  end for
  for all q ∈ body(Q) do
    b ← buckets(q)
    sortBucket(buckets,b) {MergeSort with key (#covered buckets,#views subgoals)}
  end for
```

The mapping τ relates view variables to query variables as stated in Definition 9.

The *sortBucket(buckets, b, q)* procedure decreasingly sorts the views of bucket b according to the number of covered subgoals. Views covering the same number of subgoals are sorted decreasingly according to their number of subgoals. Intuitively, this second sort criterion prioritizes the more selective views, reducing the size of the global schema instance. The sorting is implemented as a classical *MergeSort* algorithm with a complexity of $O(|M| \times log(|M|)$.

Proposition 1. *The complexity of Algorithm 1 is Max(*$O(N \times |M| \times P)$*,*$O(N \times |M| \times log(|M|))$*) where N is the number of query subgoals, M is the set of views and P is the maximal number of view subgoals.*

To illustrate Algorithm 1, consider the SPARQL query Q and the previously defined views v1-v5.

Algorithm 1 creates a bucket for each subgoal in Q as shown in Table 3a.

For instance, the bucket of subgoal $vendor(O, V)$ contains v3, v4 and v5: all the views having a subgoal covering $vendor(O, V)$. The final output after executing the *sortBucket* procedure is described in Table 3b.

Table 3: For query Q, buckets produced by Algorithm 1 when k views have been included. V_k is obtained by Algorithm 2 and the number of covered rewritings

(a) Unsorted buckets

vendor(O,V)	label(V,L)	product(O,P)	productfeature(P,F)
v3(P,L,O,R,V)	v1(P,L,T,F)	v3(P,L,O,R,V)	v1(P,L,T,F)
v4(P,O,R,V,L,U,H)	v2(P,R,L,B,F)	v4(P,O,R,V,L,U,H)	v2(P,R,L,B,F)
v5(O,V,L,C)	v3(P,L,O,R,V)		
	v4(P,O,R,V,L,U,H)		
	v5(O,V,L,C)		

(b) Sorted buckets

vendor(O,V)	label(V,L)	product(O,P)	productfeature(P,F)
v4(P,O,R,V,L,U,H)	v4(P,O,R,V,L,U,H)	v4(P,O,R,V,L,U,H)	v2(P,R,L,B,F)
v3(P,L,O,R,V)	v3(P,L,O,R,V)	v3(P,L,O,R,V)	v1(P,L,T,F)
v5(O,V,L,C)	v2(P,R,L,B,F)		
	v1(P,L,T,F)		
	v5(O,V,L,C)		

(c) Included views

# Included views (k)	Included views (V_k)	# Covered rewritings
1	v4	$1 \times 1 \times 1 \times 0 = 0$
2	v4, v2	$1 \times 2 \times 1 \times 1 = 2$
3	v4, v2, v3	$2 \times 3 \times 2 \times 1 = 12$
4	v4, v2, v3, v1	$2 \times 4 \times 2 \times 2 = 32$
5	v4, v2, v3, v1, v5	$3 \times 5 \times 2 \times 2 = 60$

Views v3 and v4 cover three subgoals, but since v4 definition has more subgoals, i.e., it is more selective, v4 is placed before v3 in all the buckets.

3.2 Global Schema Instance Construction and Query Execution

Each bucket is considered as a stack of views, having on the top the view that covers more query subgoals. A global schema instance is constructed as described in Algorithm 2 by iteratively popping one view from each bucket and loading its data into the instance.

Table 3c shows how the number of covered rewritings increases as views are included into the global schema instance. Each V_k in this table is a solution to the MaxCov problem, i.e., the number of covered rewritings for each V_k is maximal. There are two possible options regarding query execution. Query can be executed each time a new view is included into the schema instance and partial results will be produced incrementally; or, it can be executed after including the k views. The first option prioritizes the time for obtaining the first answer, while the second one favors the total time to receive all the answers of Q over

V_k. The first option produces results as soon as possible; however, in case of non-monotonic queries, i.e., queries where partial results may not be part of the query answer, this query processing approach should not be applied. Among non-monotonic queries, there are queries with modifiers like `SORT BY` or constraints like a `FILTER` that includes the negation of a `bound` expression. The execution of non-monotonic queries requires all the relevant views to be included in the global schema instance in order to produce the correct results.

Algorithm 2 The Global Schema Instance Construction and Query Execution

Input: Q : Query
Input: $Buckets$: Predicate → List<View> {The buckets are produced by Algorithm 1}
Input: k : Int
Output: A: Set<Answer>
  $Stacks$: Predicate → Stack<View>
  V_k : Set<View>
  G : RDFGraph
  for all $p \in domain(Buckets)$ **do**
    $Stacks(p) \leftarrow toStack(Buckets(p))$
  end for
  while $(\exists p| : \neg empty(Stacks(p))) \wedge |V_k| < k$ **do**
    for all $p \in domain(Stacks) \wedge \neg empty(Stacks(p))$ **do**
      $v \leftarrow pop(Stack(p))$
      if $v \notin V_k$ **then**
        load v into G {only if is not redundant}
        $A \leftarrow A \cup exec(Q, G)$ {Option 1: Execute Q after each successful load}
        $V_k \leftarrow V_k \cup \{v\}$
      end if
    end for
  end while
  $A \leftarrow exec(Q, G)$ {Option 2: execute before exit}

Proposition 2. *Considering conjunctive queries, the time complexity of Algorithm 2 in option 1 is $O(k \times N \times I)$, while the time complexity is $O(N \times I)$ for option 2. Where k is the number of relevant views included in the instance, N the number of query subgoals, and I is the size of the constructed global schema instance.*

Proposition 3. *Algorithm 2 finds a solution to the MaxCov problem.*

Proof. By contradiction, suppose that the set V_k is not maximal in terms of the number of covered rewritings, then there is another set V'_k of size k that covers more rewritings than V_k. By construction, V_k includes the first views of each bucket, i.e., the views that cover more query subgoals. There should exist at least one view in V_k that is not in V'_k, and vice-versa. Suppose w is the first view in V_k that is not in V'_k ($w \in V_k \wedge w \notin V'_k$) , v is the first one in V'_k and is not in V_k ($v \in V'_k \wedge v \notin V_k$) , and w belongs to the bucket of the query subgoal q. If v covers q, then it belongs to the bucket of q. Because V_k includes the views that cover more subgoals, if v was not included in V_k is because it covers less rewritings than w; thus, the contribution of v to the number of covered rewritings is inferior to the contribution of w. This generalizes to all the views in V'_k and not in V_k;

thus, the number of rewritings covered by V_k' should be less than the number of rewritings covered by V_k. If v covers another query subgoal q' and all the query subgoals are covered at least once by views in V_k; thus, Algorithm 2 should have included it before including w and v should belong to V_k.

3.3 The SemLAV Properties

Given a SPARQL query Q over a global schema G, a set M of views over G, the set RV of views in M relevant for Q, a set R of conjunctive queries whose union is a maximally-contained rewriting of Q using M, and V_k a solution to the MaxCov problem produced by SemLAV.

- *Answer Completeness:* If SemLAV executes Q over a global schema instance I that includes all the data collected from views in RV, then it produces the complete answer. SemLAV outputs the same answers as a traditional rewriting-based query processing approach:

$$\bigcup_{r \in R} r(I(M)) = Q(\bigcup_{v \in RV} I(v)). \tag{3}$$

- *Effectiveness:* the *Effectiveness* of SemLAV is proportional to the number of covered rewritings, it is defined as:

$$Effectiveness(V_k) = \frac{|Coverage(V_k, R)|}{|R|}. \tag{4}$$

  For an execution constrained by time or space, V_k could be smaller than RV.
- *Execution Time depends on $|RV|$:* The load and execution time of SemLAV linearly depends on the size of the views included in the global schema instance.
- *No memory blocking:* SemLAV guarantees to obtain a complete answer when $\bigcup_{v \in RV} I(v)$ fits into memory. If not, it is necessary to divide the set RV of relevant views into several subsets RV_i, such that each subset fits into memory and for any rewriting $r \in R$ all views $v \in body(r)$ are contained in one of these subsets.

4 Experimental Evaluation

We compare the SemLAV approach with a traditional rewriting-based approach and analyze the SemLAV effectiveness, memory consumption and throughput. In order to decide which rewriting engine will be use to compare with SemLAV, we run some preliminary experiments to compare existing state-of-the-art rewriting engines. We consider GQR [4], MCDSAT [14], MiniCon [15], and SSDSAT [16]. We execute these engines for 10 minutes and measure execution time and the number of rewritings generated by each engine. Additionally, we use these values to compute the throughput; throughput corresponds to number of answers

Table 4: Queries and their answer size, number of subgoals, and views size

(a) Query information

Query	Answer Size	# Subgoals
Q1	6.68E+07	5
Q2	5.99E+05	12
Q4	2.87E+02	2
Q5	5.64E+05	4
Q6	1.97E+05	3
Q8	5.64E+05	3
Q9	2.82E+04	1
Q10	2.99E+06	3
Q11	2.99E+06	2
Q12	5.99E+05	4
Q13	5.99E+05	2
Q14	5.64E+05	3
Q15	2.82E+05	5
Q16	2.82E+05	3
Q17	1.97E+05	2
Q18	5.64E+05	4

(b) Views size

Views	Size
V1-V34	201,250
V35-V68	153,523
V69-V102	53,370
V103-V136	26,572
V137-V170	5,402
V171-V204	66,047
V205-V238	40,146
V239-V272	113,756
V273-V306	24,891
V307-V340	11,594
V341-V374	5,402
V375-V408	5,402
V409-V442	78,594
V443-V476	99,237
V477-V510	1,087,281

obtained per second. Time is expressed in seconds; the total number of rewritings is computed for each query. Table 5 reports on all these metrics. The GQR performance is very good when the number of query rewritings is low, and it outperforms all the other engines. It also performs pretty well when the number of query rewritings is relatively low and views can cover more than a query subgoal. That is, this situation allows to speeds up the preprocessing time consumed by GQR to build the structures required to generate the query rewritings. The MCDSAT performance is good in a larger number of queries; it can produce rewritings for more queries than the other engines, particularly in queries which a large number of triple patterns and in presence of general predicates. However, MCDSAT does not outperform the others engines when they are able to produce the rewritings. This is because, there is an overhead in translating the problem into a logical theory which is solved using a SAT solver. The MiniCon performance is pretty good in general, but it only produces query rewritings when the space of rewritings is relatively small. Finally, SSDSAT is able to handle constants; however, this feature severely impacts its performance, being able to produce rewritings only for simple cases.

4.1 Hypothesis of Our Experimentations

The hypotheses of our experimentation are:

Table 5: Comparison of state-of-the-art LAV rewriting engines for 16 queries without existential variables and nine (plus five) views defined in [9]. The five additional views allows to cover all the queries subgoals

Query	Metric	GQR	MCDSAT	MiniCon	SSDSAT	Total Number of Rewritings
Q1	Execution Time (secs)	600.00	600.00	600.00	600.00	
	Number of Rewritings	0	247,304	0	0	2.04E+10
	Throughput (answers/sec)	0.00	412.17	0.00	0.00	
Q2	Execution Time (secs)	600.00	600.00	600.00	600.00	
	Number of Rewritings	0	0	0	0	1.57E+24
	Throughput (answers/sec)	0.00	0.00	0.00	0.00	
Q4	Execution Time (secs)	25.71	84.20	5.47	600.00	
	Number of Rewritings	16,184	16,184	16,184	0	1.62E+04
	Throughput (answers/sec)	629.38	192.22	2,957.60	0.00	
Q5	Execution Time (secs)	600.00	600.00	600.00	600.00	
	Number of Rewritings	0	513,629	0	0	7.48E+07
	Throughput (answers/sec)	0.00	856.05	0.00	0.00	
Q6	Execution Time (secs)	600.00	251.51	430.10	600.00	
	Number of Rewritings	0	314,432	314,432	0	3.14E+05
	Throughput (answers/sec)	0.00	1,250.18	731.07	0.00	
Q8	Execution Time (secs)	555.49	191.69	142.63	600.00	
	Number of Rewritings	157,216	157,216	157,216	0	1.57E+05
	Throughput (answers/sec)	283.02	820.16	1,102.30	0.00	
Q9	Execution Time (secs)	0.88	32.24	0.34	49.83	
	Number of Rewritings	34	34	34	34	3.40E+01
	Throughput (answers/sec)	38.51	1.05	101.49	0.68	
Q10	Execution Time (secs)	600.00	600.00	600.00	600.00	
	Number of Rewritings	0	656,140	0	0	4.40E+06
	Throughput (answers/sec)	0.00	1,093.57	0.00	0.00	
Q11	Execution Time (secs)	12.99	67.03	2.06	600.00	
	Number of Rewritings	9,248	9,248	9,248	0	9.25E+03
	Throughput (answers/sec)	712.15	137.96	4,487.14	0.00	
Q12	Execution Time (secs)	600.00	600.00	600.00	600.00	
	Number of Rewritings	0	440,059	0	0	1.50E+09
	Throughput (answers/sec)	0.00	733.43	0.00	0.00	
Q13	Execution Time (secs)	600.00	98.43	22.40	600.00	
	Number of Rewritings	0	64,736	64,736	0	6.47E+04
	Throughput (answers/sec)	0.00	657.69	2,890.52	0.00	
Q14	Execution Time (secs)	600.00	600.00	600.00	600.00	
	Number of Rewritings	0	913,807	0	0	2.52E+06
	Throughput (answers/sec)	0.00	1,523.01	0.00	0.00	
Q15	Execution Time (secs)	600.00	600.00	600.00	600.00	
	Number of Rewritings	0	308,903	0	0	2.04E+10
	Throughput (answers/sec)	0.00	514.84	0.00	0.00	
Q16	Execution Time (secs)	600.00	233.47	380.81	600.00	
	Number of Rewritings	0	314,432	314,432	0	3.14E+05
	Throughput (answers/sec)	0.00	1,346.81	825.68	0.00	
Q17	Execution Time (secs)	3.97	67.25	1.29	600.00	
	Number of Rewritings	4,624	4,624	4,624	0	4.62E+03
	Throughput (answers/sec)	1,165.62	68.76	3,576.18	0.00	
Q18	Execution Time (secs)	600.00	600.00	600.00	600.00	
	Number of Rewritings	0	463,754	0	0	1.20E+09
	Throughput (answers/sec)	0.00	772.92	0.00	0.00	

– SemLAV loads the more relevant views of a query first, the SemLAV effectiveness should be considerably high and should produce more answers than the rest of the engines in the same amount of time.
– SemLAV builds a global schema instance using data collected from the relevant views, SemLAV may consume more space than a traditional rewriting-based approach.
– SemLAV produces results incrementally, it is able to produce answers sooner than a traditional rewriting-based approach.

4.2 Experimental Configuration

The Berlin SPARQL Benchmark (BSBM) [8] is used to generate a dataset of 10,000,736 triples using a scale factor of 28,211 products. Additionally, third-party queries and views are used to provide an unbiased evaluation of our approach. In our experiments, the goal is to study SemLAV as a solution to the MaxCov problem, and we compute the number of rewritings generated by three state-of-the-art query rewriters. From the 18 queries and 10 views defined in [9], we leave out the ones using constants (literals) because the state-of-the-art query rewriters are unable to handle constants either in the query or in the views. In total, we use 16 out of 18 queries and nine out of 10 the defined views. The query triple patterns can be grouped into chained connected star-shaped sub-queries, that have between one and twelve subgoals with only distinguished variables, i.e., queries are free of existential variable. We define five additional views to cover all the predicates in the queries. From these 14 views, we produce 476 views by horizontally partitioning each original view into 34 parts, such that each part produces 1/34 of the answers given by the original view.

Queries and views are described in Tables 4a and 4b. The size of the complete answer is computed by including all the views into an RDF-Store (Jena) and executing the queries against this centralized RDF dataset. Query definitions are included in Appendix A.

We implement wrappers as simple file readers. For executing rewritings, we use one named graph per subgoal as done in [17]. The Jena 2.7.4[7] library with main memory setup is used to store and query the graphs. The SemLAV algorithms are implemented in Java, using different threads for bucket construction, view inclusion and query execution to improve performance. The implementation is available in the project website [8].

4.3 Experimental Results

The analysis of our results focus on three main aspects: the SemLAV effectiveness, memory consumption and throughput.

To demonstrate the SemLAV effectiveness, we execute SemLAV with a timeout of 10 minutes. During this execution, the SemLAV algorithms select and

[7] http://jena.apache.org/
[8] `https://sites.google.com/site/semanticlav/`

Table 6: The SemLAV Effectiveness. For 10 minutes of execution, we report the number of relevant views included in the global schema instance, the number of covered rewritings and the achieved effectiveness. Also values for total number of views and rewritings are shown

Query	Included Views	# Relevant Views	# Covered rewritings	# Rewritings	Effectiveness
Q1	30	408	2.28E+06	2.04E+10	0.000112
Q2	194	408	2.05E+23	1.57E+24	0.130135
Q4	156	374	8.77E+03	1.62E+04	**0.542017**
Q5	52	374	3.13E+06	7.48E+07	0.041770
Q6	44	136	2.13E+04	3.14E+05	0.067728
Q8	81	136	9.36E+04	1.57E+05	**0.595588**
Q9	34	34	3.40E+01	3.40E+01	**1.000000**
Q10	88	408	3.20E+05	4.40E+06	0.072766
Q11	77	136	5.24E+03	9.25E+03	**0.566176**
Q12	238	408	7.70E+08	1.50E+09	**0.514286**
Q13	245	408	4.26E+04	6.47E+04	**0.657563**
Q14	46	272	1.22E+04	2.52E+06	0.004837
Q15	70	442	5.12E+08	2.04E+10	0.025144
Q16	82	136	1.90E+05	3.14E+05	**0.602941**
Q17	56	136	1.90E+03	4.62E+03	0.411765
Q18	23	374	2.80E+05	1.20E+09	0.000234

include a subset of the relevant views; this set corresponds to V_k as a solution to the MaxCov problem. Then, we use these views to compute the number of covered rewritings using the formula given in Section 3. Table 6 shows the number of relevant views considered by SemLAV, the covered rewritings and the achieved effectiveness. Effectiveness is greater than or equal to 0.5 (out of 1) for almost half of the queries. SemLAV maximizes the number of covered rewritings by considering views that cover more subgoals first.

The observed results confirm that the SemLAV effectiveness is considerably high. Effectiveness depends on the number of relevant views, but this number is bounded to the number of relevant views that can be stored in memory. As expected, the SemLAV approach could require more space than the traditional rewriting-based approach. SemLAV builds a global schema instance that includes all the relevant views in V_k, whereas a traditional rewriting-based approach includes only the views in one rewriting at the time. Table 7 shows the maximal graph size in both approaches. SemLAV can use up to 129 times more memory than the traditional rewriting-based approach (for Q17). SemLAV can use less memory than the traditional rewriting-based approach (for Q1) for relevant views with overlapped data.

We calculate the throughput as the number of answers divided by the total execution time. For SemLAV, this time includes view selection and ranking, contacting data sources using the wrappers, including data into the global schema instance, and query execution time. For the traditional rewriting-based approach, this time includes rewriting time, instead of view selection and ranking. Table 7 shows for each query: number of answers, execution time, number of times the query is executed and throughput. Notice that SemLAV executes the query

Table 7: Execution of Queries Q1, Q2, Q4-Q6, Q8-Q18 using SemLAV, MCDSAT, GQR and MiniCon, using 20GB of RAM and a timeout of 10 minutes. It is reported the number of answers obtained, wrapper time (WT), graph creation time (GCT), plan execution time (PET), total time (TT), time of first answer (TFA), number of times original query is executed (#EQ), maximal graph size (MGS) in terms of number of triples and throughput (number of answers obtained per millisecond)

Query	Approach	Answer		Time (msecs)					#EQ	MGS	Throughput
		Size	%	WT	GCT	PET	TT	TFA			(answers / msec)
Q1	SemLAV	22,660,216	33	45,434	8,322	547,310	606,697	**6,370**	15	810,638	**37.3501**
	MCDSAT	290	0	13,688	202	299,546	609,381	309,952		810,409	0.0005
	GQR	0	0	0	0	0	600,415	>600,000		0	0.0000
	MiniCon	0	0	0	0	0	600,136	>600,000		0	0.0000
Q2	SemLAV	590,000	98	177,020	30,676	392,439	600,656	**260,333**	66	1,040,373	**0.9823**
	MCDSAT	0	0	15,519	105	7,058	681,246	>600,000		848,276	0.0000
	GQR	0	0	0	0	0	654,483	>600,000		0	0.0000
	MiniCon	0	0	0	0	0	600,054	>600,000		0	0.0000
Q4	SemLAV	287	**100**	555,528	73,771	327	660,938	**104,501**	47	3,659,707	**0.0004**
	MCDSAT	0	0	154,451	371	181,387	601,590	>600,000		279,896	0.0000
	GQR	0	0	557,125	1,181	11,784	600,665	>600,000		84,046	0.0000
	MiniCon	0	0	413,871	650	91,136	601,750	>600,000		177,838	0.0000
Q5	SemLAV	564,220	**100**	523,084	65,333	44,102	632,809	**116,037**	28	3,396,134	**0.8916**
	MCDSAT	0	0	398,517	384	26,287	601,731	>600,000		424,431	0.0000
	GQR	0	0	0	0	0	600,481	>600,000		0	0.0000
	MiniCon	0	0	0	0	0	600,132	>600,000		0	0.0000
Q6	SemLAV	118,258	59	547,763	62,896	13,291	625,173	**43,306**	24	2,931,316	**0.1892**
	MCDSAT	5,776	2	401,026	1,029	55,684	601,678	105,752		91,900	0.0096
	GQR	0	0	0	0	0	600,510	>600,000		0	0.0000
	MiniCon	3,697	1	193,817	248	51,300	637,514	418,169		2,184,680	0.0058
Q8	SemLAV	564,220	**100**	428,745	66,383	132,373	627,612	**5,393**	42	4,489,016	**0.8990**
	MCDSAT	16,595	2	403,133	576	65,935	603,297	113,211		256,382	0.0275
	GQR	1,706	0	330,065	194	31,587	607,594	272,737		1,264,385	0.0028
	MiniCon	467	0	198,384	349	271,398	616,114	166,776		1,265,295	0.0008
Q9	SemLAV	28,211	**100**	2,938	697	1,338	5,107	**1,235**	18	169,839	**5.5240**
	MCDSAT	28,211	100	5,609	445	1,643	41,505	34,392		5,417	0.6797
	GQR	28,211	100	3,310	132	1,281	5,709	1,435		5,417	4.9415
	MiniCon	28,211	100	3,086	129	1,362	5,004	862		5,417	5.6377
Q10	SemLAV	2,993,175	**100**	161,047	25,659	417,234	607,841	**9,810**	44	869,340	**4.9243**
	MCDSAT	332,488	11	19,801	67	383,421	600,000	207,191		603,769	0.5541
	GQR	0	0	0	0	0	600,639	>600,000		0	0.0000
	MiniCon	0	0	0	0	0	600,138	>600,000		0	0.0000
Q11	SemLAV	2,993,175	**100**	195,950	27,442	377,255	601,042	**8,352**	43	816,308	**4.9800**
	MCDSAT	1,943,141	64	141,876	389	391,852	600,000	72,939		402,528	3.2386
	GQR	1,442,134	48	248,275	689	340,937	600,000	14,435		307,089	2.4036
	MiniCon	1,956,539	65	217,321	415	385,019	605,021	6,832		402,539	3.2338
Q12	SemLAV	598,635	**100**	258,097	41,062	303,023	609,509	**5,784**	121	1,041,369	**0.9822**
	MCDSAT	0	0	424,369	498	15,271	607,408	>600,000		509,271	0.0000
	GQR	0	0	0	0	0	600,418	>600,000		0	0.0000
	MiniCon	0	0	0	0	0	600,189	>600,000		0	0.0000
Q13	SemLAV	598,635	**100**	452,288	65,043	126,345	671,893	**183,844**	124	3,509,975	**0.8910**
	MCDSAT	0	0	250,542	312	141,728	610,452	>600,000		402,531	0.0000
	GQR	0	0	36,563	344	19,757	600,376	>600,000		31,948	0.0000
	MiniCon	0	0	143,879	625	219,882	605,727	>600,000		206,689	0.0000
Q14	SemLAV	344,885	61	544,919	58,563	32,752	636,387	**29,201**	24	2,921,646	**0.5419**
	MCDSAT	10,308	1	382,674	587	63,689	614,123	133,200		1,206,075	0.0168
	GQR	0	0	0	0	0	600,714	>600,000		0	0.0000
	MiniCon	0	0	0	0	0	600,319	>600,000		0	0.0000
Q15	SemLAV	282,110	**100**	471,609	63,548	109,762	645,172	**2,911**	37	3,255,223	**0.4373**
	MCDSAT	8,298	2	90,061	271	168,041	622,474	217,445		361,882	0.0133
	GQR	0	0	0	0	0	819,679	>600,000		0	0.0000
	MiniCon	0	0	0	0	0	600,171	>600,000		0	0.0000
Q16	SemLAV	282,110	**100**	407,107	53,611	187,986	648,826	**2,531**	46	3,356,755	**0.4348**
	MCDSAT	8,298	2	437,590	852	32,015	601,584	103,641		74,682	0.0138
	GQR	1	0	26,460	79	94	619,761	619,702		1,136,305	0.0000
	MiniCon	252	0	110,366	181	122,022	603,821	400,416		1,151,769	0.0004
Q17	SemLAV	197,112	**100**	547,255	67,857	28,783	644,090	**1,504**	32	3,002,144	**0.3060**
	MCDSAT	156,533	79	412,525	1,727	60,858	600,067	70,476		23,192	0.2609
	GQR	45,037	22	245,953	177	350,406	600,000	27,178		1,098,117	0.0751
	MiniCon	5,779	2	262,608	361	334,810	600,001	26,952		1,099,508	0.0096
Q18	SemLAV	0	0	582,334	65,083	3,543	651,094	>600,000	12	2,806,533	0.0000
	MCDSAT	0	0	256,304	257	100,820	607,091	>600,000		411,901	0.0000
	GQR	0	0	0	0	0	600,791	>600,000		0	0.0000
	MiniCon	0	0	0	0	0	600,186	>600,000		0	0.0000

whenever a new relevant view has been included in the global schema instance and the query execution thread is active.

The difference in the answer size and throughput is impressive, e.g., for Q1 SemLAV produces 37.3501 answers/msec, while the other approach produces up to 0.0005 answers/msec. This huge difference is caused by the differences between the complexity of the rewriting generation and the SemLAV view selection and ranking algorithm, and between the number of rewritings and number of relevant views. This makes possible to generate answers sooner. Column TFA of Table 7 shows the time for the first answer; TFA is impacted by executing the query as soon as possible, according to option 1 given in Algorithm 2. Only for query $Q18$ SemLAV does not produce any answer in 10 minutes. This is because the views included in the global schema instance are large (around one million triples per view) and do not contribute to the answer; consequently, almost all the execution time is spent in transferring data from the relevant views. SemLAV produces answers sooner in all the other cases. Moreover, SemLAV also achieves complete answer in 11 of 16 queries in only 10 minutes.

In summary, the results show that SemLAV is effective and efficient and produces more answers sooner than a traditional rewriting-based approach. SemLAV makes the LAV approach feasible for processing SPARQL queries.

5 State of the Art

In recent years, several approaches have been proposed for querying the Web of Data [18–22]. Some tools address the problem of choosing the sources that can be used to execute a query [21, 22]; others have developed techniques to adapt query processing to source availability [18, 21]. Finally, frameworks to retrieve and manage Linked Data have been defined [19, 21], as well as strategies for decomposing SPARQL queries against federations of endpoints [6]. All these approaches assume that queries are expressed in terms of RDF vocabularies used to describe the data in the RDF sources; thus, their main challenge is to effectively select the sources, and efficiently execute the queries on the data retrieved from the selected sources. In contrast, SemLAV attempts to integrate data sources, and relies on a global schema to describe data sources and to provide a unified interface to the users. As a consequence, in addition to collecting and processing data transferred from the selected sources, SemLAV decides which of these sources need to be contacted first, to quickly answer the query.

Three main paradigms have been proposed to integrate dissimilar data sources. In GAV mediators, entities in the global schema are semantically described using views in terms of the data sources. In consequence, including or updating data sources may require the modification of a large number of mappings [3]. In contrast, the LAV approach, new data sources can be easily integrated [3]; further, data sources that publish entities of several concepts in the global schema, can be naturally defined as LAV views. Thus, the LAV approach is best suited for applications with a stable global schema but with changing data sources; contrary, the GAV approach is more suitable for applications with stable data sources and

a changing global schema. Finally, a more general approach named Global-Local-As-View (GLAV) allows the definition of mappings where views on the global schema are mapped to views of the data sources. Recently, Knoblock et al. [23] and Taheriyan et al. [24] proposed Karma, a system to semi-automatically generate source descriptions as GLAV views on a given ontology. Karma makes GLAV views a solution to consume open data as well as to integrate and populate these sources into the LOD cloud.

GLAV views are suitable not only to describe sources, but also to provide the basis for the dynamic integration of open data and Web APIs into the LOD cloud. Further, theoretical results presented by Calvanese et al.[7] establish that for conjunctive queries against relational schemas, GLAV query processing techniques can be implemented as the combination of the resolution of the query processing tasks with respect to the LAV component of the GLAV views followed by query unfolding tasks on the GAV component. Thus, SemLAV can be easily extended to manage GLAV query processing tasks, and provides the basis to integrate existing GLAV views. Additionally, SemLAV can be used to develop SPARQL endpoints that dynamically access up-to-date data from the data sources or Web APIs defined by the generated GLAV views.

The problem of rewriting a query into queries on the data sources is a relevant problem in integration systems [25]. A great effort has been made to provide solutions able to produce query rewritings in the least time possible and to scale up to a large number of views. Several approaches have been defined, e.g., MCDSAT [14], GQR [4], Bucket Algorithm [25], and MiniCon [11]. Recently, Le et al, [17] propose a solution to identify and combine GAV SPARQL views that rewrite SPARQL queries against a global vocabulary, and Izquierdo et al [16] extend the MCDSAT rewriter with preferences to identify the combination of semantic services that rewrite a user request. Recently, Montoya et al propose GUN [26], a strategy to maximize the number of answers obtained from a given set of k rewritings; GUN aggregates the data obtained from the relevant views present in those k rewritings and executes the query over it. Even if GUN could maximize the number of obtained answers, it would still depend on query rewritings as input, and has no criteria to order the relevant views.

We address this problem and propose SemLAV, a query processing technique for RDF store architectures that provides a uniform interface to data sources that have been defined using the LAV paradigm [27]. SemLAV gets rid of the query rewriter, and focuses on selecting relevant views for each subgoal of the query. Moreover, SemLAV decides which relevant views will be contacted first, and includes the retrieved data into a global schema instance where the query is executed. At the cost of memory consumption, SemLAV is able to quickly produce answers first, and compute a *more complete* answer when the rest of the engines fail. Since the number of valid query rewritings can be exponential in the number of views, providing an effective and efficient semantic data management technique as SemLAV is a relevant contribution to the implementation of integration systems, and provides the basis for feasible and dynamic semantic integration architectures in the Web of Data.

An alternative approach for data integration is Data Warehousing [28], where data is retrieved from the sources and stored in a repository. In this context, query optimization relies on materialized views that allows to speed up the execution time. Selecting the best set of views to be materialized is a complex problem that has been deeply studied in the literature [9, 29–32]. Commonly approaches attempt to select this set of views according to an expected workload and available resources. Recently, Castillo-Espinola [9] propose an approach where materialized views correspond to indexes for SPARQL queries that allow to speed up query execution time. Although these approaches may considerably improve performance in average, only queries that can be rewritten using the materialized views will be benefited. Further, the cost of the view maintainability process can be very high if data frequently changes and it needs to be kept up-to-date to ensure answer correctness.

SemLAV also relies on view definitions, but views are temporally included in the global schema instance during query execution; thus, data is always up-to-date. Furthermore, the number of views to be considered is not limited. The only limitation depends on the physical resources available to perform a particular query. Nevertheless, it is important to highlight that the number of relevant views for answering one query is, in the general case, considerably smaller than the total number of views in the integration system.

6 Conclusions and Future Work

In this paper, we presented SemLAV, a Local-As-View mediation technique that allows to perform SPARQL queries over views without facing problems of NP-completeness, exponential number of rewritings or restriction to conjunctive SPARQL queries. This is obtained at the price of including relevant views into a global schema instance which is space consuming. However, we demonstrated that, even if only a subset of relevant views is included, we obtain more results than traditional rewriting-based techniques. Chances of producing results are higher, if the number of covered rewritings is maximized as defined in the MaxCov problem. We proved that our ranking strategy maximizes the number of covered rewritings.

SemLAV opens a new way to execute SPARQL queries for LAV mediators that is tractable. As perspectives, the performance of SemLAV can be greatly improved by parallelizing views inclusion. Currently, SemLAV includes views sequentially due to Jena restrictions. If views were included in parallel, time to get first results would be greatly improved. Additionally, the strategy of producing results as soon as possible, can deteriorate the overall throughput. If users want to improve overall throughput, then the query should be executed once after all the views in V_k have been included. It could be also interesting to design an execution strategy where SemLAV would execute under constrained space. In this case, the problem would be to find the minimum set of relevant views that would fit in the available space and produce the maximal number of answers. All these problems will be part of our future works.

Acknowledgments. We thank C. Li for providing his MiniCon code, and J. L. Ambite and G. Konstantinidis for sharing the GQR code for the evaluation. This work is partially supported by the French National Research agency (ANR) through the KolFlow project (code: ANR-10-CONTINT-025), part of the CONTINT research program, and by USB-DID.

References

1. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. Int. J. Semantic Web Inf. Syst. **5** (2009) 1–22
2. Abiteboul, S., Manolescu, I., Rigaux, P., Rousset, M.C., Senellart, P.: Web Data Management. Cambridge University Press, New York, NY, USA (2011)
3. Ullman, J.D.: Information integration using logical views. Theor. Comput. Sci. **239** (2000) 189–210
4. Konstantinidis, G., Ambite, J.L.: Scalable query rewriting: a graph-based approach. In Sellis, T.K., Miller, R.J., Kementsietsidis, A., Velegrakis, Y., eds.: SIGMOD Conference, ACM (2011) 97–108
5. Vidal, M.E., Ruckhaus, E., Lampo, T., Martínez, A., Sierra, J., Polleres, A.: Efficiently joining group patterns in sparql queries. In Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T., eds.: ESWC (1). Volume 6088 of Lecture Notes in Computer Science., Springer (2010) 228–242
6. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: Fedx: Optimization techniques for federated query processing on linked data. [33] 601–616
7. Calvanese, D., Giacomo, G.D., Lenzerini, M., Vardi, M.Y.: Query processing under glav mappings for relational and graph databases. PVLDB **6** (2012) 61–72
8. Bizer, C., Schultz, A.: The berlin sparql benchmark. Int. J. Semantic Web Inf. Syst. **5** (2009) 1–24
9. Castillo-Espinola, R.: Indexing RDF data using materialized SPARQL queries. PhD thesis, Humboldt-Universität zu Berlin (2012)
10. Wiederhold, G.: Mediators in the architecture of future information systems. IEEE Computer **25** (1992) 38–49
11. Halevy, A.Y.: Answering queries using views: A survey. VLDB J. **10** (2001) 270–294
12. Lenzerini, M.: Data integration: A theoretical perspective. In Popa, L., Abiteboul, S., Kolaitis, P.G., eds.: PODS, ACM (2002) 233–246
13. Doan, A., Halevy, A.Y., Ives, Z.G.: Principles of Data Integration. Morgan Kaufmann (2012)
14. Arvelo, Y., Bonet, B., Vidal, M.E.: Compilation of query-rewriting problems into tractable fragments of propositional logic. In: AAAI, AAAI Press (2006) 225–230
15. Pottinger, R., Halevy, A.Y.: Minicon: A scalable algorithm for answering queries using views. VLDB J. **10** (2001) 182–198
16. 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.: International Semantic Web Conference (1). Volume 6496 of Lecture Notes in Computer Science., Springer (2010) 386–401
17. Le, W., Duan, S., Kementsietsidis, A., Li, F., Wang, M.: Rewriting queries on sparql views. In Srinivasan, S., Ramamritham, K., Kumar, A., Ravindra, M.P., Bertino, E., Kumar, R., eds.: WWW, ACM (2011) 655–664

18. Acosta, M., Vidal, M.E., Lampo, T., Castillo, J., Ruckhaus, E.: Anapsid: An adaptive query processing engine for sparql endpoints. [33] 18–34
19. Basca, C., Bernstein, A.: Avalanche: Putting the spirit of the web back into semantic web querying. In Polleres, A., Chen, H., eds.: ISWC Posters&Demos. Volume 658 of CEUR Workshop Proceedings., CEUR-WS.org (2010)
20. Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K.U., Umbrich, J.: Data summaries for on-demand queries over linked data. In Rappa, M., Jones, P., Freire, J., Chakrabarti, S., eds.: WWW, ACM (2010) 411–420
21. Hartig, O.: Zero-knowledge query planning for an iterator implementation of link traversal based query execution. [34] 154–169
22. Ladwig, G., Tran, T.: Sihjoin: Querying remote and local linked data. [34] 139–153
23. Knoblock, C.A., Szekely, P.A., Ambite, J.L., Gupta, S., Goel, A., Muslea, M., Lerman, K., Mallick, P.: Interactively mapping data sources into the semantic web. In Kauppinen, T., Pouchard, L.C., Keßler, C., eds.: LISC. Volume 783 of CEUR Workshop Proceedings., CEUR-WS.org (2011)
24. Taheriyan, M., Knoblock, C.A., Szekely, P.A., Ambite, J.L.: Rapidly integrating services into the linked data cloud. In Cudré-Mauroux, P., Heflin, J., Sirin, E., Tudorache, T., Euzenat, J., Hauswirth, M., Parreira, J.X., Hendler, J., Schreiber, G., Bernstein, A., Blomqvist, E., eds.: International Semantic Web Conference (1). Volume 7649 of Lecture Notes in Computer Science., Springer (2012) 559–574
25. Levy, A.Y., Rajaraman, A., Ordille, J.J.: Querying heterogeneous information sources using source descriptions. In Vijayaraman, T.M., Buchmann, A.P., Mohan, C., Sarda, N.L., eds.: VLDB, Morgan Kaufmann (1996) 251–262
26. Montoya, G., Ibáñez, L.D., Skaf-Molli, H., Molli, P., Vidal, M.E.: Gun: An efficient execution strategy for querying the web of data. In Decker, H., Lhotská, L., Link, S., Basl, J., Tjoa, A.M., eds.: DEXA (1). Volume 8055 of Lecture Notes in Computer Science., Springer (2013) 180–194
27. Levy, A.Y., Mendelzon, A.O., Sagiv, Y., Srivastava, D.: Answering queries using views. In Yannakakis, M., ed.: PODS, ACM Press (1995) 95–104
28. Theodoratos, D., Sellis, T.K.: Data warehouse configuration. In Jarke, M., Carey, M.J., Dittrich, K.R., Lochovsky, F.H., Loucopoulos, P., Jeusfeld, M.A., eds.: VLDB, Morgan Kaufmann (1997) 126–135
29. Gupta, H.: Selection of views to materialize in a data warehouse. In Afrati, F.N., Kolaitis, P.G., eds.: ICDT. Volume 1186 of Lecture Notes in Computer Science., Springer (1997) 98–112
30. Chirkova, R., Halevy, A.Y., Suciu, D.: A formal perspective on the view selection problem. VLDB J. **11** (2002) 216–237
31. Karloff, H.J., Mihail, M.: On the complexity of the view-selection problem. In Vianu, V., Papadimitriou, C.H., eds.: PODS, ACM Press (1999) 167–173
32. Goasdoué, F., Karanasos, K., Leblay, J., Manolescu, I.: View selection in semantic web databases. PVLDB **5** (2011) 97–108
33. Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N.F., Blomqvist, E., eds.: The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I. In Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N.F., Blomqvist, E., eds.: International Semantic Web Conference (1). Volume 7031 of Lecture Notes in Computer Science., Springer (2011)
34. Antoniou, G., Grobelnik, M., Simperl, E.P.B., Parsia, B., Plexousakis, D., Leenheer, P.D., Pan, J.Z., eds.: The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May

29-June 2, 2011, Proceedings, Part I. In Antoniou, G., Grobelnik, M., Simperl, E.P.B., Parsia, B., Plexousakis, D., Leenheer, P.D., Pan, J.Z., eds.: ESWC (1). Volume 6643 of Lecture Notes in Computer Science., Springer (2011)

A Queries

In our experimental study, we evaluate the SPARQL queries proposed by Castillo-Espinola [9]. We only consider the SPARQL queries without constants or literals due to limitations of state-of-the-art rewriters.

```
SELECT *
WHERE {
    ?X1 rdfs:label ?X2 .
    ?X1 rdf:type ?X3 .
    ?X1 bsbm:productFeature ?X4 .
    ?X1 bsbm:productFeature ?X5 .
    ?X1 bsbm:productPropertyNumeric1 ?X6 .
}
```

Listing 1.6: Q1

```
SELECT *
WHERE {
    ?X1 rdfs:label ?X2 .
    ?X1 rdfs:comment ?X3 .
    ?X1 bsbm:producer ?X4 .
    ?X4 rdfs:label ?X5 .
    ?X1 dc:publisher ?X4 .
    ?X1 bsbm:productFeature ?X6 .
    ?X6 rdfs:label ?X7 .
    ?X1 bsbm:productPropertyTextual1 ?X8 .
    ?X1 bsbm:productPropertyTextual2 ?X9 .
    ?X1 bsbm:productPropertyTextual3 ?X10 .
    ?X1 bsbm:productPropertyNumeric1 ?X11 .
    ?X1 bsbm:productPropertyNumeric2 ?X12 .
}
```

Listing 1.7: Q2

```
SELECT *
WHERE {
    ?X1 rdfs:label ?X2 .
    ?X1 foaf:homepage ?X3 .
}
```

Listing 1.8: Q4

```
SELECT *
WHERE {
    ?X1 bsbm:vendor ?X2 .
    ?X1 bsbm:offerWebpage ?X3 .
    ?X2 rdfs:label ?X4 .
    ?X2 foaf:homepage ?X5 .
}
```

Listing 1.9: Q5

```
SELECT *
WHERE {
    ?X1 bsbm:reviewFor ?X2 .
    ?X1 rev:reviewer ?X3 .
    ?X1 bsbm:rating1 ?X4 .
}
```

Listing 1.10: Q6

```
SELECT *
WHERE {
    ?X1 bsbm:offerWebpage ?X2 .
    ?X1 bsbm:price ?X3 .
    ?X1 bsbm:deliveryDays ?X4 .
}
```

Listing 1.11: Q8

```
SELECT *
WHERE {
    ?X1 bsbm:productPropertyNumeric1 ?X2 .
}
```

Listing 1.12: Q9

```
SELECT *
WHERE {
    ?X1 rdfs:label ?X2 .
    ?X1 rdf:type ?X3 .
    ?X1 bsbm:productFeature ?X4 .
}
```

Listing 1.13: Q10

```
SELECT *
WHERE {
    ?X1 rdf:type ?X2 .
    ?X1 bsbm:productFeature ?X3 .
}
```

Listing 1.14: Q11

```
SELECT *
WHERE {
    ?X1 bsbm:producer ?X2 .
    ?X2 rdfs:label ?X3 .
    ?X1 dc:publisher ?X2 .
    ?X1 bsbm:productFeature ?X4 .
}
```

Listing 1.15: Q12

```
SELECT *
WHERE {
    ?X1 bsbm:productFeature ?X2 .
    ?X2 rdfs:label ?X3 .
}
```

Listing 1.16: Q13

```
SELECT *
WHERE {
    ?X1 bsbm:producer ?X2 .
    ?X3 bsbm:product ?X1 .
    ?X3 bsbm:vendor ?X4 .
}
```

Listing 1.17: Q14

```
SELECT *
WHERE {
    ?X1 rdfs:label ?X2 .
    ?X3 bsbm:reviewFor ?X1 .
    ?X3 rev:reviewer ?X4 .
    ?X4 foaf:name ?X5 .
    ?X3 dc:title ?X6 .
}
```

Listing 1.18: Q15

```
SELECT *
WHERE {
    ?X1 bsbm:reviewFor ?X2 .
    ?X1 dc:title ?X3 .
    ?X1 rev:text ?X4 .
}
```

Listing 1.19: Q16

```
SELECT *
WHERE {
    ?X1 bsbm:reviewFor ?X2 .
    ?X1 bsbm:rating1 ?X3 .
}
```

Listing 1.20: Q17

```
SELECT *
WHERE {
    ?X1 bsbm:product ?X2 .
    ?X2 rdfs:label ?X3 .
    ?X1 bsbm:vendor ?X4 .
    ?X1 bsbm:price ?X5 .
}
```

Listing 1.21: Q18