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

Abstract. Local-as-view mediators allow to query semantic heterogeneous data sources such as linked data endpoints, open data servers or any available web service. Nevertheless, the problem of rewriting conjunctive queries on LAV mediators is intractable in general, and the number of rewritings can be exponential on the size of the query. Most important, in the context of the Semantic Web even simple conjunctive SPARQL queries can be rewritten in millions of rewritings. In this paper, we propose SemLAV, a LAV mediation technique that relies on Semantic Web technologies to execute SPARQL queries against data sources. SemLAV implements a polynomial view selection algorithm to choose the relevant views that can answer a given SPARQL query. Additionally, SemLAV implements an execution strategy for gathering data from the relevant views and outputs a great portion of the answer in a short time. We present an experimental study for SemLAV, and compare its performance with execution strategies for LAV rewritings. The results suggest that SemLAV scales up to SPARQL queries and a large number of views, while it significantly outperforms existing solutions.

1 Introduction

SPARQL federation engines attempt to execute queries against federations of SPARQL endpoints. Although the great effort done by the community to make datasets accessible by SPARQL endpoints, there are still a large number of open data sources and services that do not support the SPARQL protocol and cannot be integrated by existing federation engines. Nevertheless, traditional data integration approaches could be used to perform SPARQL queries over heterogeneous data sources, and thus integrate and provide a uniform interface to SPARQL endpoints, datasets and Web services.

Local-as-view (LAV) mediation [1] is a well-known and flexible approach to perform data integration over heterogeneous and autonomous data sources. A LAV mediator relies on views to define semantic mappings between an uniform interface defined at the mediator level, and local schemas or views that describe the integrated data sources. A LAV mediator relies on a query rewriter to translate a mediator query into the union of queries against the local views. LAV is suitable for environments where data sources frequently change, and entities of different types are defined in a single source, e.g., an open source that publishes data about restaurants and museums in different French cities; further, LAV can naturally integrate sources from the Web of Data [2]. However, LAV mediation has well known severe bottlenecks: (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 the case of conjunctions of triple patterns. For example, in traditional database system, a LAV mediator with 140 conjunctive views can generate 10,000 rewritings for a conjunctive query with 8 goals [3]. In contrast, the number of rewritings for a SPARQL query can be much larger. To explain, 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 published by the majority of the data sources. Additionally, these triple patterns can be grouped into chained connected star-shaped sub-queries [4]. Finally, a large number of variables can be projected out. Thus all these properties impacts on the complexity of the query rewriting problem and conduces to the explosion of the number of query rewritings. For example, a SPARQL query with 12 triple patterns that comprises three chained star-shaped sub-queries can be rewritten using 300 views in billions of rewritings, whenever some triple patterns in the query are bound to general predicates. This problem is even more challenging considering that statistics may be unavailable, and there are not clear criteria to rank or prune the generated rewritings [5].

In this paper, we propose SemLAV, the first scalable LAV-based approach for the Semantic Web. Given a SPARQL query Q on a set of views V, SemLAV avoids producing and executing rewritings by processing the original query Q over the aggregation of data gathered from relevant views. As SemLAV executes the original query, it allows: (i) executing full SPARQL queries, (ii) avoiding the query rewriting problem which is NP-complete, (iii) making the query execution influenced by the number of relevant views rather than the number of rewritings.

SemLAV builds an aggregation graph composed of data from relevant views; thus space required to build this graph may be considerably larger than the space required to execute all the query rewritings one by one. Nevertheless, evaluating the original query once on the aggregation graph can produce the answers obtained executing all the rewritings. To empirically evaluate the properties of SemLAV, we conducted an experimental study using the Berlin Benchmark [6] and queries and views designed by Castillo-Espinola [7]. Results suggest that SemLAV outperforms existing algorithms with respect to answers produced per time unit, and provides a scalable LAV-based solution to the problem of executing SPARQL queries over data sources. The contributions of this paper are: (i) The first scalable and effective LAV-based approach for the Semantic Web. (ii) Formalization of the problem of finding the set of relevant views that maximizes the number of covered rewritings (MaxCov). (iii) An optimal solution to the MaxCov problem for views without existential variables. (iv) A LAV-based query engine able to execute full SPARQL queries, and 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

Definition 1. *An RDF LAV integration system is 3-tuple IS=<MS,S,V>, where MS is a mediator RDF vocabulary or schema, S is a set of data sources, and $V = \{v_1, \ldots, v_m\}$ is a set of the LAV views that define the sources in S. Views are defined as SPARQL queries over the mediator RDF vocabulary MS. D is a virtual RDF dataset on the mediator RDF vocabulary MS.*

Definition 2. *Let IS=<MS,S,V>, $I = \{I(v_1), \ldots, I(v_m)\}$ and Q be an RDF LAV integration system, a set of instances of the views in V and a SPARQL query expressed in terms of MS, where an instance of a view, $I(v_i)$, is a subset of the result set of evaluating the SPARQL query that defines the view v_i over D, i.e., $I(v_i) \subseteq v_i(D)$. This means that data sources might be incomplete and have only some of the RDF triples from D that satisfy the view definition. The problem of computing Q over V corresponds to answer Q using the instances of the views in V.*

Although this problem has been extensively studied by the Database community [8], it has never been addressed for SPARQL queries.

In the context of conjunctive queries, some approaches have been proposed to solve this problem by proposing a solution to the Query Rewriting Problem using views (QRP)[3, 9]. Thus query rewriters have been developed to rewrite a conjunctive query over the mediator schema into query rewritings on the views [10, 11]. In general, a query rewriter performs two steps: view selection and query rewriting. During the view selection step, the query rewriter identifies the set of relevant views for each query subgoal q in the body of Q. A view v is relevant to a query subgoal q, if q is part of the subgoals that define the view v.

Definition 3. *The set of relevant views for a subgoal $q \in body(Q)$ is defined as: $RV(V, q) = \{v : v \in V \wedge covers(v, q)\}$, we say that* view v covers subgoal q *if there exists a mapping τ from $Vars(Q) \rightarrow Vars(V)$ such that $\tau(q) = g$ for g a view subgoal of v.*

Once relevant views are chosen, the rewriter algorithm combines these views to produce the query rewritings.

Definition 4. *A rewriting is a conjunctive query $r(\bar{x})$:- $v_1(\bar{x_1}), \ldots, v_m(\bar{x_n})$ where $v_i \in V$ and v_i is a relevant view. The body of each query rewriting contains a relevant view that covers each subgoal of Q.*

A query rewriting must be *contained* in Q, i.e., for all dataset D and a set of views V over D, the result set of executing r in V is contained in the result set of executing Q over D, i.e., $r(I(v_1), \ldots, I(v_n)) \subseteq Q(D)$. Existing solutions are tailored to find the maximally-contained rewriting.

Maximally Contained Query Rewriting Problem (QRP). Given a conjunctive query Q and a set of views V = { $v_1, \ldots, v_n$ } over a dataset D, QRP is to find a set of rewritings R, called the solution of the QRP, such that:

– For all instances of the views in the bodies of all rewritings in R, the union of the results of executing each query rewriting is contained in the result of executing Q in D, i.e., $\bigcup_{r \in R} r(I(v_1), \ldots, I(v_n)) \subseteq Q(D)$
– R is maximal, i.e., there is no other set R', such that:

$$\bigcup_{r \in R} r(I(v_1), \ldots, I(v_n)) \subset \bigcup_{r' \in R'} r'(I(v_1), \ldots, I(v_n)) \subseteq Q(D)$$

Theorem 1. *Let N, M and V be the number of query subgoals, the maximal number of views subgoals, and the set of views, the number of candidate rewritings in the worst case is:* $(M \times |V|)^N$ *[2].*

Theorem 2. *Time complexity of QRP is NP-Complete [8].*

We illustrate the limitations of the LAV-based query rewriting approach for SPARQL queries through an example. In the following example, the global schema is defined over the Berlin Benchmark [6] vocabulary. Consider two SPARQL queries $Q1$ and $Q2$ [7] on our global schema; $Q1$ has 7 subgoals and provides different information about products as shown in Listing 1.1. Further, $Q2$ has 12 subgoals and retrieves information about products, producers and publishers as show in Listing 1.2.

```
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 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 1.2: SPARQL Q2

Consider 14 data sources defined as conjunctive views over the global schema as in Listing 1.3; the Berlin Benchmark [6] vocabulary terms are represented as binary predicates in the conjunctive queries that define the data sources.

For instance, $s1$ retrieves information about *product type*, *label* and *product feature*. The *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. Thus the general predicate `rdfs:label` in queries $Q1$ and $Q2$ can be mapped to views *s1, s3-s7, s11-17*. To illustrate how the number of rewritings for $Q1$ and $Q2$ can be affected by the number of data sources that publish the general predicate `rdfs:label`, we use the LAV query rewriter MCDSAT[3] [9]. First, if 14 data sources are considered, $Q1$

[3] MCDSAT [9] is only one publicly available that outputs the number of rewritings without enumerating all of them.

can be rewritten in 42 rewritings. For 28 data sources, there are 5,376 rewritings, and 1.12743e+10 rewritings are produced for 224 sources. Because $Q2$ has a larger number of triple patterns and three of them are bound to the general predicate `rdfs:label`, we obtain 658,560 rewritings for 14 data sources, and similarly to $Q1$ if more data sources are considered, the number of rewritings grows exponentially. For 28 sources, MCDSAT computes 2.69746e+09 rewritings, while 1.85368e+20 rewritings are calculated for 224 views. Just with two simple queries, 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.

```
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 1.3: Views s1-s10 from [7]

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

Query	Rewriter	5 minutes	10 minutes	20 minutes
Q1	GQR	0	0	0
	MCDSAT	211,125	440,308	898,766
	MiniCon	0	0	0
Q2	GQR	0	0	0
	MCDSAT	67,028	157,909	335,063
	MiniCon	0	0	0

Table 1: Number of rewritings obtained from rewriters GQR, MCDSAT and MiniCon with timeouts of 5, 10 and 20 minutes. Using 224 views and queries Q1 and Q2.

rewritings obtained by state-of-the-art LAV rewriters GQR[3], MCDSAT[9] and MiniCon[11] when 224 views are considered for queries $Q1$ and $Q2$ and timeouts are set up to 5, 10 and 20 minutes. Note that all these engines are able to produce only empty results or a small number of rewritings.

Finally, because general predicates correspond to terms used to define properties of resources of any domain, views defined in terms of these predicates do not necessarily contribute to answer queries that comprise triple patterns bound to them, e.g., the general predicate `rdfs:label` in view *s7* is used to describe *reviews* and not *producers* as in $Q2$. Thus, although all the query rewritings were computed for $Q2$, the performance of the query engine would be affected by the execution of useless query rewritings, as the ones that comprise view *s7*. In summary, even if the LAV approach constitutes a flexible approach to make data from heterogeneous data sources available, query rewriting and processing tasks may be unfeasible in the context of the Semantic Web. 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 the Semantic Web, we propose a novel approach that identifies a query relevant views, and by executing the original query over the data gathered from the relevant views, outputs a high percentage of the answer in short time.

3 The SemLAV Approach

SemLAV is a LAV approach for the Semantic Web. It considers that no statistics about the data sources are available due to their highly dynamic nature. SemLAV follows the traditional Mediator and Wrappers architecture [12]. Given SPARQL query Q over a set of views $V = \{ v_1, \ldots, v_n \}$ defined over a dataset D, SemLAV avoids producing and executing rewritings by running the original query Q over the aggregation of data gathered from relevant views defined in V. In order to make the original query executable, SemLAV needs special wrappers. Traditional wrappers return the instantiation of the head of views, SemLAV wrappers return the instantiation of the body of views. For example, if the relevant view $s1$ defined as $s1(X1, X2, X3, X4) :- label(X1, X2), type(X1, X3), productfeature(X1, X4)$ is part of a rewriting to be executed, the $s1$ wrapper will produce tuples with values for variables $X1, X2, X3, X4$ in relation $s1(X1,\ X2,\ X3,\ X4)$. SemLAV wrappers will produce a graph defined by the body of $s1$ with values for variables, i.e., $X1\, rdfs : label\, X2\ .\ X1\, rdf : type\, X3\ .\ X1\, bsbm : productFeature\, X4$. SemLAV wrappers could be more expensive in space than traditional wrappers, especially in presence of existential variables in views. However it allows to execute full SPARQL queries and makes the query execution dependent on the number of relevant views and no more on the number of rewritings. As stated before, in the context of SPARQL queries and the Semantic Web, the number of relevant views is lower by an exponential factor.

If all relevant views for query Q can be aggregated, SemLAV produces complete answers. However, the number of relevant views could be considerably large. If we only have resources to consider k relevant views, V_k, we should consider the ones that increase the chance of obtaining answers. We define the chance of obtaining answers is proportional to the number of rewritings that V_k covers, i.e., the number of rewritings that only use views present in V_k. Moreover, the order in which views are aggregated has an impact in answers' generation time, then the views should be considered accordingly to their chance of producing answers.

Maximal Coverage Problem (MaxCov). Given $k > 0$, Q a query defined for dataset D, V a set of views over dataset D, R solution of QRP for Q and V. Let V_k be subset of V of relevant views for Q of size k. Find V_k, such that the set of rewritings covered by V_k, $MaxCov(V_k, R)$, is maximal for all subsets of V of size k. $MaxCov(V_k, R)$ is defined as:

$$MaxCov(V_k, R) = \{r : r \in R \wedge (\forall p : p \in \mathit{body}(r) : p \in V_k)\}$$

Let Q be $Q(\bar{X})$:- $p_1(\bar{X}_1), \ldots p_n(\bar{X}_n)$, the number of rewritings covered by V_k in the worst case is: $|MaxCov(V_k, R)| = |RV(V_k, p_1)| \times \ldots |RV(V_k, p_n)|$, with RV as stated in Definition 3, $RV(V_k, p_i) = \{v : v \in V_k \wedge covers(v, p_i)\}$.

This corresponds to the number of combinations that could be generated by the V_k relevant views, with one view per query subgoal. And it coincides with the number of rewriting for queries without existential variables. Let $R_k \subseteq R$ be the subset of rewritings covered by V_k, the following expression holds:

$$Q(I(V_k)) = \bigcup_{r \in R_k} r(I(V_r)) \subseteq Q(D)$$

The execution of the original query over the union of the relevant views V_k is equivalent to the execution of the rewritings $r \in R_k$ over the instance of views in r. This is contained in the execution of the original query over the dataset D. If $MaxCov(V_k, R)$ is equal to R then the complete answer will be generated.

For the rest of the paper, we consider that views contain only distinguished variables, which correspond to the worst case in terms of number of rewritings. This assumption allow us to compute the number of covered rewritings for Q using only views in V_k as $|RV(V_k, p_1)| \times |RV(V_k, p_2)| \ldots |RV(V_k, p_n)|$. Otherwise, this formula is just an upper bound.

3.1 SemLAV Relevant Views Selection and Ranking

The relevant views selection and ranking algorithm finds the views that can cover each of the query subgoals. This algorithm is similar to the first step of Bucket algorithm[8]. It creates a bucket for each query subgoal q. The main difference is that our algorithm sorts the Buckets' views according to the number of subgoals they cover. Hence, the views that are more likely to contribute to the answer will be considered first. The view selection and ranking algorithm is defined in Algorithm 1.

The mapping τ relates the variables in a predicate of a view to the variables in a subgoal of Q, it corresponds to the identity mapping[4] for query variables and a valid transformation for query constants. Let upper case letters be variables and lower case be constants, if the query contains $p1(X, Y)$, $p1(Y, W)$ and $p1(W, z)$, and some view $v1(A, B, C)$ contains $p1(A, B)$, then $v1(A, B, C)$ and $v1(A, z, C)$ should be included in the buckets. Since the buckets will not be used to generate rewritings there is not need to consider $v1(X, Y, \_0)$ and $v1(Y, W, \_1)$ as a rewriting algorithm could do. If such a mapping exists between a predicate in a view v and a subgoal q of Q, we can add the

[4] The identity mapping Id, is defined as Id(X) = X, for all values of X.

Algorithm 1 Relevant views selection and ranking

```
Require: $Q$ : SPARQL Query
Require: $V$: Set of Views defined as conjunctive queries
Ensure: $Buckets$: Predicate $\rightarrow$ List<View>
  for all $q \in body(Q)$ do
      $buckets(q) \leftarrow \emptyset$
  end for
  for all $q \in body(Q)$ do
      $b \leftarrow buckets(q)$
      for all $v \in V$ do
          for all $w \in body(v)$ do
              Let $\tau$ be a mapping from $Vars(w)$ to $Vars(w) \bigcup Constants$
              if $\tau$ exists then
                  $v_i \leftarrow \lambda(v)$ {$\lambda(v)$ replaces all variables $a_i$ in the head of $v$ by $\tau(a_i)$}
                  $insert(b, vi)$ {add $v_i$ to the bucket if is not redundant}
              end if
          end for
      end for
  end for
  for all $q \in body(Q)$ do
      $b \leftarrow buckets(q)$
      sortBucket(buckets, b, q)
  end for
```

view result of applying τ to the variables in the head of v (represented in the algorithm by the λ function) to the subgoal bucket if it is not redundant. That is, the bucket does not contain an occurrence of this view whose instantiation contains the instantiation of v, e.g., if the view $v(X, Y)$ with X and Y variables is already in the bucket, the view $v(X, c)$ with c constant will not be added as it is contained in $v(X, Y)$.

The $sortBucket(buckets, b, q)$ procedure sorts decreasingly the views of bucket b by the number of subgoals they cover. Views covering the same number of subgoals are sorted increasingly according to their number of subgoals. Intuitively, this second sort criteria prioritizes the more selective views, reducing the size of the loaded dataset. The sorting is implemented as a classical *MergeSort* algorithm with a complexity of $O(|V| \times log(|V|)$.

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

To illustrate Algorithm 1, consider a query with 4 subgoals:
Q(O,V,L,P,F) :- vendor(O, V), label(V,L), product(O,P), productfeature(P,F)
and 5 data sources:

```
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)
```

Algorithm 1 creates a bucket for each subgoal in Q as shown in Table 2a.

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 2b.

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)		

(a) Unsorted 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)		

(b) Sorted buckets

# 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	$3 \times 5 \times 2 \times 2 = 60$

(c) Included views

Table 2: For query Q, buckets produced by Algorithm 1, and included views when including only k views as done by Algorithm 2 and their number of covered rewritings.

The $v4$ and $v3$ views cover three subgoals, but since $v4$ definition has more subgoals, i.e., it is more selective, $v4$ is in first place in buckets that contain it. The next step is to use these views to build the dataset for executing Q.

3.2 Dataset Construction and Query Execution

Once the views have been selected and ranked according to their coverage, they are included in a dataset as described in Algorithm 2. Each bucket is considered as a stack of views, having on the top the view that covers more query subgoals. The dataset is constructed by iteratively popping one view from each bucket and aggregating it in a dataset.

Table 2c shows the included views as the number of included views increases. All V_k shown is a solution to the MaxCov problem, i.e., the number of covered rewritings is maximal. There are two options regarding when the query is executed: one, execute it each time a new view is included in a dataset; two, execute it at the end of the execution. The first option corresponds to a setup where we prioritize the time for obtaining the first answer; the second one prioritizes the total time to get all the answers of Q over V_k.

Proposition 2. *The complexity of Algorithm 2 is $O(k \times VI)$, where k is the number of included views and VI is the size of the largest view instance.*

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

Algorithm 2 Dataset Construction and Query Evaluation

```
Require: $Q$ : Query
Require: $Buckets$: Predicate → List<View> {The buckets are produced by Algorithm 1}
Require: $k$ : Int
Ensure: $A$: 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 $p \in domain(Stacks)$ 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}
```

Proof. By contradiction, suppose the constructed set V_k is not maximal in terms of covered rewritings, then there is another set V'_k of size k that covers more rewritings than V_k, let r be one of the rewritings that V'_k covers and V_k does not. r should include a view that does not belong to V_k, and since this view was not included then it should cover the same or less subgoals than the included views. If it covers the same number of query subgoals, then it does not contribute to cover more rewritings because to include it, another view that covers the same or more subgoals should be not considered, then it does not appears in r. If it covers less query subgoals, then including it instead of any included view decreases the number of covered subgoals, then V'_k does not cover more rewritings than V_k.

3.3 SemLAV's Properties

We state some properties of SemLAV and compare it with rewriting based approaches. Given a SPARQL query Q, a set of views V on a dataset D, RV the set of relevant views for Q and R a solution to the QRP of Q over V.

- *Answer Completeness:* If SemLAV executes Q over a dataset instance D_i that includes all data gathered from views in RV, then it will produce the complete answer, i.e., SemLAV will produce the same answers as rewriting based approaches: $\bigcup_{r \in R} r(I(V)) = Q(\bigcup_{v \in RV} I(v))$.
- *Effectiveness:* We define the *Effectiveness* of SemLAV as:
  $Effectiveness(V_k) = \frac{|MaxCov(V_k,R)|}{|R|}$.
  Where V_k corresponds to the set of relevant views included in the dataset instance D_i. 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 loaded into the aggregation graph.

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

(a) Query 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

(b) Views size

Table 3: Queries and their answer size and number of subgoals, and views size.

– *No memory blocking:* SemLAV guarantees to obtain complete answer when $\bigcup_{v \in RV} I(v)$ fits in memory. If not, it is necessary to divide this set of relevant views RV into several subsets RVi such that each subset fits into the memory and that for any $r \in R$ all views $v \in body(r)$ are contained in one RVi. It is possible to obtain an approximate solution, following an heuristic that considers only a subset of RVis.

4 Experimental Evaluation

We compare SemLAV approach with rewriting based approaches and analyse SemLAV's effectiveness, memory consumption and throughput. We report results with MCDSAT rewriter since it has better performance in terms of number of rewritings generated per time unit for the considered setup[5]. We evaluate only conjunctive queries because MCDSAT is limited to these queries.

4.1 Experimental setup

We used the Berlin SPARQL Benchmark (BSBM) [6] to generate a dataset of 10,000,736 triples using a scale factor of 28,211 products. We used a third part queries and views to have no biased evaluation of our approach. We used the 16 of 18 queries and the 9 of 10 views defined in [7] that do not use constants because MCDSAT does not handle constants. Queries triple patterns can be grouped into chained connected star-shaped sub-queries, that have between 1 and 12 subgoals, with only distinguished variables. We defined 5 additional views to cover all the predicates in the queries. From these 14 views, we produced 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.

[5] Rewriting generation results with different rewriters are available in the project website: https://sites.google.com/site/semanticlav/

Query	Included Views	# Relevant Views	# Rws covered	# Rewritings	Effectiveness
Q1	28	408	1.61E+06	2.04E+10	0.000079
Q2	194	408	2.05E+23	1.57E+24	0.130135
Q4	56	374	1.90E+03	1.62E+04	0.117647
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	80	408	2.60E+05	4.40E+06	0.059045
Q11	77	136	5.24E+03	9.25E+03	**0.566176**
Q12	232	408	7.06E+08	1.50E+09	0.471565
Q13	126	408	1.30E+04	6.47E+04	0.200754
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

Table 4: SemLAV's Effectiveness. For 10 minutes of execution, we report the number of relevant views included in the dataset, the number of covered rewritings and the achieved effectiveness. Also values for total number of views and rewritings are shown.

Queries and views information is shown in Tables 3a and 3b. The size of the complete answer was computed by loading all the views into a RDF-Store (Jena) and executing the queries over it.

We implemented wrappers as simple file readers. For executing rewritings, we used one named graph per subgoal as done in [13]. The Jena 2.7.4 [6] library with main memory setup was used to store and query the graphs. SemLAV algorithms were implemented in Java, using different threads for bucket construction, view inclusion and query execution to improve performance.

4.2 Experimental Results

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

To demonstrate SemLAV's effectiveness, we executed SemLAV with a timeout of 10 minutes. During this execution SemLAV algorithms selected and included a subset of relevant views, this set corresponds to V_k as solution to the MaxCov problem. Then, we use these views to calculate the number of covered rewritings using the formula given in section 3. Table 4 shows the number of relevant views considered by SemLAV, the covered rewritings and the effectiveness achieved. The achieved effectiveness is greater or equal than 0.5 out of 1 for some queries. The set of gathered relevant views covers a large number of rewritings, i.e., all the rewritings using only views in this set. SemLAV strategy of considering views that cover more subgoals first maximizes the number of covered rewritings.

The observed results confirms that SemLAV effectiveness is considerably high. Effectiveness depends on the number of relevant views considered, 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 rewriting based approach. SemLAV builds an aggregation dataset that includes all relevant views, whereas a query

[6] http://jena.apache.org/

Query	Approach	Answer		Time (msecs)					#EQ	MGS	Throughput
		Size	%	WT	GCT	PET	TT	TFA			(answers / sec)
Q1	SemLAV	22,246,301	33	52,256	9,195	542,987	604,286	**5,902**	15	801,627	**36.8142**
	MCDSAT	317	0	12,504	62	311,662	600,680	289,030		810,409	0.0005
Q2	SemLAV	590,000	98	177,020	30,676	392,439	600,656	**260,333**	66	1,040,373	**0.9823**
	MCDSAT	0	0	0	0	0	600,000	>600,000		0	0.0000
Q4	SemLAV	287	**100**	560,677	70,965	301	632,062	**115,906**	31	3,412,246	**0.0005**
	MCDSAT	0	0	155,351	174	189,524	612,940	>600,000		279,896	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	405,759	344	25,687	602,473	>600,000		424,431	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	403,348	811	56,610	601,086	100,452		91,900	0.0096
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	402,012	449	66,506	601,117	107,382		256,382	0.0276
Q9	SemLAV	28,211	**100**	2,938	697	1,338	5,107	**1,235**	18	169,839	**5.5240**
	MCDSAT	28,211	**100**	3,310	124	1,314	37,069	32,774		5,417	0.7610
Q10	SemLAV	2,993,175	**100**	171,013	25,337	404,418	600,989	**8,065**	43	824,834	**4.9804**
	MCDSAT	228,349	7	9,572	46	256,702	448,288	191,471		603,768	0.5094
Q11	SemLAV	2,993,175	**100**	195,950	27,442	377,255	601,042	**8,352**	43	816,308	**4.9800**
	MCDSAT	1,907,730	63	91,753	96	264,353	420,335	71,618		402,513	4.5386
Q12	SemLAV	598,635	**100**	234,632	39,574	328,225	602,828	**3,068**	119	1,036,395	**0.9930**
	MCDSAT	0	0	424,818	325	15,730	601,307	>600,000		509,271	0.0000
Q13	SemLAV	598,635	**100**	488,922	67,080	78,313	634,433	**157,221**	67	3,527,436	**0.9436**
	MCDSAT	0	0	248,692	193	142,694	607,291	>600,000		402,531	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	397,880	425	62,664	624,110	130,015		1,206,146	0.0165
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	89,588	183	175,862	625,689	204,674		361,882	0.0133
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	437,477	747	33,289	600,928	100,343		74,682	0.0138
Q17	SemLAV	197,112	**100**	547,255	67,857	28,783	644,090	**1,504**	32	3,002,144	**0.3060**
	MCDSAT	150,736	76	431,549	1,587	52,683	600,374	67,644		23,192	0.2511
Q18	SemLAV	0	0	582,334	65,083	3,543	651,094	>600,000	12	2,806,533	0.0000
	MCDSAT	0	0	255,801	256	101,093	602,413	>600,000		411,901	0.0000

Table 5: Execution of Queries Q1, Q2, Q4-Q6, Q8-Q18 using SemLAV and MCDSAT, 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) and 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).

rewriter approach includes only the views in a rewriting. Table 5 shows the maximal graph size in both approaches. SemLAV can use up to 129 times more memory than the rewriting based approach (for Q17), but it is also possible that SemLAV uses less memory than the rewriting based approach (for Q1) for sources with overlapped data. This additional memory enhances SemLAV's performance in terms of throughput.

We calculated 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, loading data into the aggregated graph, and plan execution time. For MCDSAT, this time includes rewriting time, instead of view selection and ranking. Table 5 shows the queries' execution time, number of answers, throughput and number of times original queries are executed. Notice that we execute the original queries if a new relevant view has been included and the executing thread is active.

The difference in answers' size and throughput is impressive, i.e., for Q1 SemLAV produces 36,831 answers/sec, while the other approach produces 0 answers/sec. The reasons of this huge difference most likely are the difference in complexity of the rewriting generation and SemLAV's view selection and ranking, and the difference between the number of rewritings and number of relevant views. This makes possible to generate more answers sooner. Column TFA of Table 5 shows the time of first answer, only for query $Q18$ SemLAV does not produce any answer in 10 minutes because the views included in the dataset are big (around 1 million triples per view) but not contributing to results. Consequently, most execution time is spent in data transferring from sources. In all the other cases SemLAV produces answers sooner, this value also is impacted by executing the original query as often as possible, according to option 1 given in Algorithm 2. Moreover, SemLAV also achieved 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 the traditional LAV query rewriting approaches. SemLAV makes LAV approach feasible for the Semantic Web.

5 State of the art

In recent years, several approaches have been proposed for querying the Web of Data [14–18]. Some tools address the problem of choosing the sources that can be used to execute a query [17, 18]; others have developed techniques to adapt query processing to source availability [14, 17]. Finally, frameworks to retrieve and manage Linked Data have been defined [15, 17], as well as strategies for decomposing SPARQL queries against federations of endpoints [5]. 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 our approach attempts to semantically integrate data sources, and relies on a global vocabulary to describe data sources and provide a unified interface to the users. As a consequence, in addition to collecting and processing data transferred from the selected sources, it decides which of these sources need to be contacted first, to quickly answer the query.

Two main paradigms have been proposed to integrate dissimilar data sources. In GAV mediators, entities in the RDF global vocabulary 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 [19]. In contrast, the LAV approach, new data sources can be easily integrated [19]; further, data sources that publish entities of several concepts in the RDF global vocabulary, can be naturally defined as LAV views. Thus the LAV approach is best suited for applications with a stable RDF global vocabulary but with changing data sources whereas the GAV approach is best suited for applications with stable data sources and a changing vocabulary. Given the nature of the Semantic Web, we rely on the LAV approach to describe data sources as views over a global RDF vocabulary, and assume that the global vocabulary of concepts is stable while data sources may constantly pop up or disappear from the Web.

The problem of rewriting a global query into queries on the data sources is one relevant problem in integration systems [10], and several approaches have been defined to efficiently enumerate the query rewritings and to scale when a large number of views exists (e.g., MCDSAT [9], GQR [3], Bucket Algorithm [10], MiniCon [8]). Recently, Le et al, [13] propose a solution to identify and combine GAV SPARQL views that rewrite SPARQL queries against a global vocabulary, and Izquierdo et al [20] extends the MCDSAT with preferences to identify the combination of semantic services that rewrite a user request. A great effort has been made to provide solutions able to produce query rewritings in the least time possible. Recently, Montoya et al propose GUN [21], an 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 original query over it. Even if GUN maximizes the number of answers obtained, it still depends on query rewritings as input, and has no criteria to select the order in which views are aggregated.

We address this problem and propose SemLAV, a query processing technique for RDF store architectures that provide a uniform interface to data sources that have been defined using the LAV paradigm [1]. SemLAV gets rid of the query rewriters, and focuses on selecting relevant views for each subgoal of the input query. Moreover, SemLAV decides which relevant sources will be contacted first, and gathers the retrieved data into an aggregated graph where the input query is executed. At the cost of memory consumption, SemLAV is able to quickly produce first answers, 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 sources, 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 semantic integration architectures in the Web of Data.

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 aggregating relevant views, which is space consuming. However, we demonstrated that, even if only a subset of relevant views is aggregated, as it covers an exponential number of rewritings, we obtain more results than traditional techniques.The chance of getting results is higher if the number of covered rewritings is maximized as defined in the MaxCov problem. We demonstrated that our ranking strategy maximized the number of covered rewritings.

SemLAV opens a new way to execute queries for LAV mediators that is tractable in the context of SPARQL queries. As perspectives, the performance of SemLAV can be greatly improved by parallelizing views aggregation. Currently, SemLAV load views sequentially due to Jena restriction. If views are loaded in parallel, time to get first results may be greatly improved. Additionally, the strategy of producing results as soon as possible, can degrade overall throughput. If the user wants to improve overall throughput, then the original query should be executed once after all views in V_k have been loaded. It can be also interesting to design an execution strategy where SemLAV executes under constrained space. In this case, the problem is to find the minimum set of relevant views that fits in the available space and produces the maximal number of answers.

References

1. Levy, A.Y., Mendelzon, A.O., Sagiv, Y., Srivastava, D.: Answering queries using views. In: Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'95. (1995) 95–104
2. Abiteboul, S., Manolescu, I., Rigaux, P., Rousset, M., Senellart, P.: Web data management. Cambridge University Press (2011)
3. Konstantinidis, G., Ambite, J.L.: Scalable query rewriting: a graph-based approach. In: SIGMOD Conference. (2011) 97–108
4. Vidal, M.E., Ruckhaus, E., Lampo, T., Martinez, A., Sierra, J., Polleres, A.: Efficiently Joining Group Patterns in SPARQL Queries. In: ESWC. (2010)
5. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: Fedx: Optimization techniques for federated query processing on linked data. In: ISWC. (2011) 601–616
6. Bizer, C., Shultz, A.: The berlin sparql benchmark. International Journal on Semantic Web and Information Systems **5** (2009) 1–24
7. Castillo-Espinola, R.: Indexing RDF data using materialized SPARQL queries. PhD thesis, Humboldt-Universität zu Berlin (2012)
8. Halevy, A.Y.: Answering queries using views: A survey. The VLDB Journal **10** (2001) 270–294
9. Arvelo, Y., Bonet, B., Vidal, M.E.: Compilation of query-rewriting problems into tractable fragments of propositional logic. In: AAAI. (2006) 225–230
10. Levy, A., Rajaraman, A., Ordille, J.: Querying heterogeneous information sources using source descriptions. In: VLDB. (1996) 251–262
11. Pottinger, R., Levy, A.Y.: Minicon: A scalable algorithm for answering queries using views. In: VLDB. (2000) 484–495
12. Wiederhold, G.: Mediators in the architecture of future information systems. IEEE Computer **25** (1992) 38–49
13. Le, W., Duan, S., Kementsietsidis, A., Li, F., Wang, M.: Rewriting queries on sparql views. In: WWW, ACM (2011) 655–664
14. Acosta, M., Vidal, M.E., Lampo, T., Castillo, J., Ruckhaus, E.: Anapsid: An adaptive query processing engine for sparql endpoints. In: ISWC (1). (2011) 18–34
15. Basca, C., Bernstein, A.: Avalanche: Putting the Spirit of the Web back into Semantic Web Querying. In: SSWS. (2010) 64–79
16. Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K.U., Umbrich, J.: Data summaries for on-demand queries over linked data. In: WWW. (2010) 411–420
17. Hartig, O.: Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In: ESWC. (2011) 154–169
18. Ladwig, G., Tran, T.: Sihjoin: Querying remote and local linked data. In: ESWC. (2011) 139–153
19. Ullman, J.D.: Information integration using logical views. Theoretical Computer Science **239** (2000) 189–210
20. Izquierdo, D., Vidal, M.E., Bonet, B.: An expressive and efficient solution to the service selection problem. In: ESWC. (2010) 386–401
21. Montoya, G., Ibanez, L.D., Skaf-Molli, H., Molli, P., Vidal, M.E.: GUN: An Efficient Execution Strategy for Querying the Web of Data. Technical report, Nantes University (2013)