RDF Pattern Matching using Sortable Views

Zhihong Chong
Southeast University
SiPaiLou #2, Nanjing, China, 210096
chongzhihong@seu.edu.cn

He Chen
Southeast University
SiPaiLou #2, Nanjing, China, 210096
hchen.seu@gmail.com

Zhenjie Zhang
Advanced Digital Sciences Center
Illinois at Singapore Pte. Ltd., Singapore
zhenjie@adsc.com.sg

Hu Shu
Southeast University
SiPaiLou #2, Nanjing, China, 210096
hshu.seu@gmail.com

Guilin Qi
Southeast University
SiPaiLou #2, Nanjing, China, 210096
qiguiling@seu.edu.cn

Aoying Zhou
East China Normal University
3663 Zhongshan Rd (North), Shanghai, China, 200062
ayzhou@sei.ecnu.edu.cn

ABSTRACT

In the last few years, RDF is becoming the dominating data model used in *semantic web* for knowledge representation and inference. In this paper, we revisit the problem of pattern matching query in RDF model, which is usually expensive in efficiency due to the huge cost on join operations. To alleviate the efficiency pain, view materialization techniques are usually deployed to accelerate the query processing. However, given an arbitrary view, it remains difficult to identify how to reuse the view for a particular query, because of the NP-hardness behind the algorithm matching patterns and views. To fully exploit the benefit of the materialized views, we propose a new paradigm to enhance the effectiveness of the materialized view. Instead of choosing materialized views in arbitrary form, our paradigm aims to select the views *only if* they are *sortable*. The property of *sortability* raises huge gains on the pattern-view matching, bringing down the cost to linear complexity in terms of the pattern size. On the other side, the costs on identifying sortable views and searching over the views using inverted index are affordable. Moreover, sortable views generally improve the overall performance of pattern matching, by means of a cost model used to optimize the query rewriting on the most appropriate views. Finally, we demonstrate extensive experimental results to verify the superiority of our proposal on both efficiency and effectiveness.

Categories and Subject Descriptors: H.2.4 [Database Management]: Query processing

Keywords: RDF Query, RDF Indexing, sortable view.

1. INTRODUCTION

In the last few years, RDF is becoming the dominating data model used in *semantic web* for knowledge representation and inference [3, 11], because of its advantages on describing heterogenous and unstructured data. Table 1 is an example of RDF graph G, representing relationships among a group of persons. The records in the table, for example, show that 1) *Alice* marries *Bob*; 2) *Alice* is *Chris*'s mother; 3) *Chris* coauthored with *Bob* and *Frank*, etc. The evaluation of RDF queries usually resorts to the pattern matching methods in the standard SPARQL language[3]. In Figure 1, we present an example pattern V_1, by which the user is able to find father-son pairs coauthoring on the same papers. As shown in figure, a variable begins with prefix ? and a labeled edge represents a triple pattern.

subject	predicate	object
Alice	*Marry*	*Bob*
Alice	*isMother*	*Chris*
Chris	*coAuthor*	*Bob*
Chris	*coAuther*	*Frank*
Alice	*isMother*	*David*
...	...	...

Table 1: An Example RDF Data Graph

In Table 2, we list all valid results to query V_1, in which the row with $?x = Alice$, $?y = Bob$ and $?z = Chris$ implies that Bob is father of $Chris$ and they have coauthored before. This simple example illustrates the advantages of RDF on flexible data representation and powerful knowledge inference. However, answering such queries typically involves a long chain of join operations. The cost of *join proliferation* becomes an efficiency bottleneck[20].

Materialization techniques are usually deployed on RDF models to alleviate the difficulty of pattern matching. Assume that another query Q_1 shown in Figure 1 is submitted. Although it looks very different from V_1, one can easily verify the existence of a mapping, i.e., $\phi_1 = \{?x \mapsto ?a, ?y \mapsto ?c, ?z \mapsto ?b\}$, to transform V_1 to Q_1', which is a sub-query/subgraph of Q_1.

The mapping above, known as *containment mapping* [17], implies that the results of V_1 in Table 2 could be recycled to find all matchings to Q_1', by renaming $?x$, $?y$ and $?z$ with $?a$, $?c$ and $?b$, respectively. Therefore, the RDF engine is capable of deriving the complete answer to Q_1 by joining the results of Q_1' and $Q_1 - Q_1'$. This evaluation procedure only involves a single join operation, which is much more efficient than the query processing scheme answering the query completely from scratch.

Despite of the huge performance gain using materialization ap-

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.

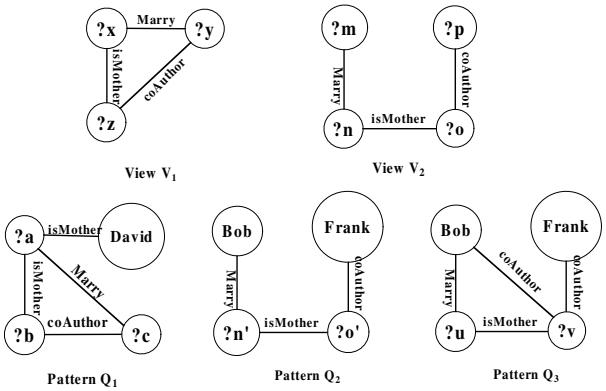


Figure 1: Running Example

V_1	?x	?y	?z
Instance $[V_1]$	*Alice*	*Bob*	*Chris*
	...	...	...

Table 2: Materialize View $[V_1]_G$ for V_1

proaches [17, 15], there remain technical challenges in query rewriting with respect to the materialized views. Given a pattern query for matching and an existing materialized view, it is generally NP-hard to find the containment mapping between the query and the view [17], due to the equivalence between the matching problem and subgraph isomorphism problem [23, 21]. This means the complexity of finding a containment mapping is exponential to the number of variables and references in the worst case, unless P=NP. In this paper, we explore on a new direction to fully utilize the advantage of materialized views without the overhead on containment mapping identification. Specifically, instead of selecting arbitrary views for materialization, our approach aims to select the views *only if* they are *sortable*, namely there is a unique order on the nodes in the pattern graph of the views.

Sortable views are generally attractive on a couple of different aspects. Firstly, given a sortable materialized view, there exists a linear algorithm to recover the containment mapping between the view and the query pattern, even when the query pattern is not *sortable*. This property enables fast verification on the reusability of the materialized views on *any* pattern query matching. Secondly, the construction/selection of sortable views does not incur significant costs. From a conventional RDF view, a simple topological ordering on the graph nodes is sufficient for a reconstruction following the requirements on sortability. Thirdly, traditional inverted index structure seamlessly supports fast retrieval on potential sortable views, when a large pool of candidate views are available to pattern matching queries. We show that a particular view is subject to certain containment mapping to part of the query, only when every edge in the sortable view has a unique matching to the query edge, after careful query expansion. Last but not least, the sortable view technique leads to a simple optimization model, which facilitates the system to find the combination of views with minimal expected cost on query processing. All these features make our sortable views as an enticing enhancement on top of the existing materialized view methods in RDF engines. We hereby summarize the main contributions of the paper as follows.

1. We revisit the problem of answering queries using views in the context of RDF model and raise the problem of simplifying pattern rewriting via reasonably restricting view selection.
2. We identify a special class of views, called sortable views, for restricting view selection, which potentially provides huge gain in pattern rewriting.
3. We present efficient index for query rewriting and optimization model to prune redundant views.
4. We implement a prototype system using the view selection method based on sortable views, which could be seamlessly integrated with traditional relational databases.

The remainder of the paper is organized as follows. Section 2 reviews existing studies on RDF query processing and graph matching. Section 3 introduces the problem definitions and preliminaries. Section 4 presents sortable views and analyzes its properties. Section 5 proposes the pattern rewriting techniques and discuss how to support the rewriting with inverted index. Section 6 covers the experimental results and finally Section 7 concludes this paper.

2. RELATED WORK

The problem of join proliferation in RDF query has received considerable attention [20, 9]. The current practices mainly resort to speeding up operations by either index [1, 14, 20] or optimization techniques [20, 22]. In this paper, we aims to reduce join operations using views. Our work differs from flat storage schema [9] for this purpose in that it does not demand a prior defined storage schema and therefore is more flexible.

The problem of reformulating queries using views for query optimization or database design dates back to equivalent conjunctive queries in relational data model [17, 5], and were expanded to encompass recursive queries [13] or even queries with aggregation functions [10] and later revisited in semi-structural data model with regular path queries [16]. In addition to theoretical interests on complexity [2], minimization [8] of reformulation of queries and other constraints [19] in the presence of materialized views, practical and scalable solutions to exploiting views are intensively studied, including [6]. In this paper, we focus on the view exploitation [17] and particularly answer which views are usable for a given RDF pattern, covering view selection in view design and rewriting for view exploitation [6].

While addressed in relational [5] and XML models [16], it yet raises additional complexity for controlling join proliferation [20, 3, 11] in RDF query evaluation. On one hand, views defined by patterns in RDF model syntactically resemble those defined by conjunctive queries in relational model. On the other hand, data in RDF model are grouped together in one table other than distributed over several flat tables in relational model. This seemly minor difference introduces additional difficulties. Different table names involved in relational conjunctive queries help reduce the number of candidate containment mappings, because containment mapping must relate the same table names from view and query, respectively. In a similar way, tree-structured XML pattern helps reduce the difficulty of XML rewriting [16], compared with graph-structured patterns in RDF model.

Our solution to RDF pattern rewriting using views is different from the recent work about RDF views [15, 12, 4], targeted at simplifying pattern rewriting. The view selection defined in [15] is thoroughly different from our sortable view selection, optimizing a candidate view set for given queries. Thus, our work is orthogonal to the view selection in [15]. Although, the works in [12, 4] touche on which views to be possibly involved in query optimization, our work clearly departs from them, dedicated to simplifying pattern rewriting via sortable view selection; [12] studies shortcuts especially for RDF path queries while [4] focusing on cost model for

view selection. Therefore, our work can be considered as a generalization of shortcuts in [12] and a different effort from [4].

Serialization for matching also appears in other work [21], targeted at reducing the space of candidate matchings. However, our solution technique identifies itself in that it produces a mapping rather than reducing searching space; an order is defined to sort triple patterns in view and serialize those in query. This sorting and serialization produces a containment mapping which is a kind of matching.

3. PROBLEM DEFINITIONS AND PRELIMINARIES

3.1 Pattern Matching

We use $\mathbb{D}_u$ and $\mathbb{D}_v$ to denote the disjoint domains of URI references (simply references) and variables respectively. Let $\mathbb{D} = \mathbb{D}_u \cup \mathbb{D}_v$. Note that blank nodes are not considered as explained in [3]. Triple space is $\mathbb{T} = \mathbb{D}_u \times \mathbb{D}_u \times \mathbb{D}_u$ and Triple pattern space is $\mathbb{P} = \mathbb{D} \times \mathbb{D} \times \mathbb{D}$. Given a triple pattern q, $q[1]$ denotes the subject of q. Similarly, $q[2]$ and $q[3]$ stand for the predicate and object, respectively.

Data graph G is a set of triples in $\mathbb{T}$, i.e. $G \subseteq \mathbb{T}$. Query pattern Q (simply pattern) is a set of triple patterns, i.e. $Q = \{q_1, q_2, \ldots, q_n\}$, where $q_i \in \mathbb{P}$ for $1 \leq i \leq n$. They are used to represent the conjunctive semantics of triple patterns. For a pattern Q, $\mathsf{var}(Q)$, $\mathsf{uri}(Q)$ denote the sets of variables and references in Q, respectively. Let $\mathsf{dom}(Q) = \mathsf{var}(Q) \cup \mathsf{uri}(Q)$.

A matching μ on pattern Q with respect to data graph G is defined as $\mu : \mathsf{var}(Q) \rightarrow \mathbb{D}_u$, transforming a pattern Q to a graph $\mu(Q)$ by replacing every variable $?x \in \mathsf{var}(Q)$ with the corresponding reference $\mu(?x)$. Let $[Q]_G$ denote the set of all matchings of Q on data graph G, i.e. $[Q]_G = \{\mu | \mu(Q) \subseteq G\}$. The subscription of G can be omitted if the context is clear. Recall the example of V_1, one matching $\mu = \{?x \mapsto Alice, ?y \mapsto Bob, ?z \mapsto Chris\}$ is stored in Table 2. Essentially, $\mu(V_1)$ transforms pattern V_1 to a sub-graph.

3.2 Pattern Rewriting

View $[V]$ is the materialized matchings for pattern V. Pattern V is also called the definition of view $[V]$. Therefore, we interchangeably use a view and its definition without ambiguity. The containment mapping is the core component of answering queries using views [17], which concerns which views to be used and how to use them. We give a formal definition below, in the context of pattern matching for RDF query [3, 11].

DEFINITION 1. ***Containment Mapping***
Containment mapping ϕ is a partial injective mapping $\phi : \mathbb{D}_v \rightarrow \mathbb{D}$ with domain $\mathsf{dom}(\phi)$ and range $\mathsf{ran}(\phi)$.

Recall our previous example for V_1 and Q_1 in Figure 1. Containment mapping $\phi_1 = \{?x \mapsto ?a, ?y \mapsto ?c, ?z \mapsto ?b\}$ maps V_1 to Q_1', i.e., $\phi(V_1) = Q_1'$. To be precise, for triple pattern q, $\phi(q)$ is obtained by replacing $x \in \mathsf{dom}(q) \cap \mathsf{dom}(\phi)$ with y according to $x \mapsto y$. Hence, $\phi(Q) = \{\phi(q) | q \in Q\}$. For empty mapping ϕ, we define $\phi(q) = q$.

DEFINITION 2. ***Containment***
Pattern V contains Q, denoted by $V \sqsupseteq Q$, if there is a containment mapping ϕ such that $\phi(V) \subseteq Q$.

For example, $V_1 \sqsupseteq Q_1$ under containment mapping ϕ_1 in the previous example.

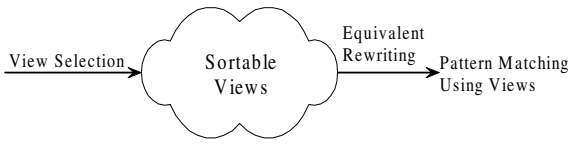


Figure 2: Framework

A view base Γ is a set of views, i.e., $\Gamma = \{V_1, V_2, \cdots, V_n\}$. The problem of pattern rewriting against it is defined as follows.

DEFINITION 3. ***Pattern Rewriting***
Given pattern Q against view base Γ, pattern rewriting is to rewrite Q into a set $\Gamma(Q)$ of patterns, i.e., $\Gamma(Q) = \{\phi_i(V_i) | V_i \in \Gamma, V_i \sqsupseteq Q\}$. Its matchings $[\Gamma(Q)]$ is defined as $[\Gamma(Q)] = \cdots \bowtie [\phi_i(V_i)] \bowtie \cdots$, where $\phi_i(V_i) \in \Gamma(Q)$.

It worths noting that $[\phi_i(V_i)]$ is considered as a table for storing matchings as shown in Table 2. Therefore, two tables $[\phi_i(V_i)]$ and $[\phi_j(V_j)]$ can be joined with relational operator $\bowtie$ on common variables(or common attributes).

Let $\Gamma = \{V_1, V_2\}$ in Figure 1, $\Gamma(Q_3) = \{\phi_{3_1}(V_1), \phi_{3_2}(V_2)\}$ since one can verify $V_1 \sqsupseteq Q_3$ and $V_2 \sqsupseteq Q_3$ under some containment mappings ϕ_{3_1} and ϕ_{3_2}, respectively.

3.3 Pattern Matching via Pattern Rewriting

In this par of the section, we discuss how to employ pattern rewriting to complete pattern matching using view V. In particular, $V \sqsupseteq \phi(V)$ under containment mapping ϕ. Let $\phi = \phi^v \cup \phi^u$, where ϕ^v maps variables to variables and ϕ^u to references.These two parts are used to determine relational renaming operator ρ_{ϕ^v} and selection operator σ_{ϕ^u}, respectively. Here, for $?x \mapsto ?y$ in ϕ^v, the renaming operator renames $?x$ with $?y$. In the similar way, for $?x \mapsto y$ in ϕ^u, select operator works with the selection of $?x = y$.

LEMMA 1. *Given two patterns V and Q with the same length, if $V \sqsupseteq Q$ under the containment mapping ϕ, we have,*

$$[Q] = [\phi(V)] = \pi_{\mathsf{var}(\phi(V))}(\rho_{\phi^v}(\sigma_{\phi^u}([V]))).$$

For example, the containment mapping $\phi_2 = \{?m \mapsto Bob, ?p \mapsto Frank, ?n \mapsto ?n', ?o \mapsto ?o'\}$ between V_2 and Q_2 in Figure 1 consists of two parts. The first part $\phi_2^v = \{?n \mapsto ?n', ?o \mapsto ?o'\}$ maps variables to variables and the second part $\phi_2^u = \{?m \mapsto Bob, ?p \mapsto Frank\}$ maps variables to references. To obtain $[Q_2]$, we could apply selection operator $\sigma_{?m=Bob,?p=Frank}$ with respect to ϕ^u on materialized view $[V_2]$, and renaming one $\rho_{?n \mapsto ?n', ?o \mapsto ?o'}$ with respect to ϕ^v. Finally, by applying projection operator $\pi_{?n',?o'}$, we get

$$[Q_2] = \pi_{?n',?o'}(\rho_{?n \mapsto ?n', ?o \mapsto ?o'}(\sigma_{?m=Bob,?p=Frank}([V_2]))).$$

Lemma 1 illustrates how to evaluate via pattern rewriting $\Gamma(Q)$. For each $\phi_i(V_i) \in \Gamma(Q)$, it can be obtained via view V_i by using basic relational operators. Let $Q - \Gamma(Q) = Q - \bigcup_{\phi_i(V_i) \in \Gamma} \phi_i(V_i)$. We summarize how pattern rewriting is used to solve pattern matching in Theorem 1.

THEOREM 1. *Given pattern Q against view base Γ, $[Q] = [\Gamma(Q)] \bowtie [Q - \Gamma(Q)]$.*

We use Figure 2 to illustrate the main idea behind Theorem 1. Generally speaking, the rewriting consists of two steps, namely

view selection and equivalent pattern rewriting [17]. In view selection, instead of using existing cost model [25], we consider view selection on behalf of rewriting. In particular, we select sortable views for materialization, because of its attractive advantages for query rewriting.

4. SORTABLE VIEW

Theorem 1 states the efficacy of pattern rewriting defined in Definition 3. The key problem involved in this theorem is how to find containment mapping ϕ_i between view V_i and Q such that $[\phi_i(V_i)]$ is evaluated in terms of Lemma 1. Consequently, $[\Gamma(Q)]$ is obtained using views. To achieve this goal, an order, called *containment order*, is introduced. The order is used to sort views and serialize patterns. The sorting and serialization greatly simplifies the searching for containment mapping.

4.1 Containment Order

The intuition behind containment order is that it preserves the equivalence between tripe patterns v and $\phi(v)$ under any containment mapping ϕ. Therefore, this order can be used as a necessary condition for containment verification because each triple pattern in view must be mapped to its equivalent patterns in query. We find lexical order suffices this goal.

DEFINITION 4. ***Containment Order on*** $\mathbb{D}$
$(\preceq, \mathbb{D})$: $\forall x, y \in \mathbb{D}_u$, $x \prec y$, iff x precedes y in terms of lexical order. In other case, $\forall x, y \in \mathbb{D}_u \cup \mathbb{D}_v$, they are equivalent, denoted by $x \simeq y$. Further, $x \preceq y$ if $x \prec y$ or $x \simeq y$.

Essentially, if $x \simeq y$, there is a containment mapping ϕ such that $\phi(x) = y$ or $\phi(y) = x$. The defined *containment order* is naturally extended to triple pattern space $\mathbb{P}$.

DEFINITION 5. ***Containment Order on*** $\mathbb{P}$
$(\preceq, \mathbb{P})$: A triple pattern q_1 is ordered before triple pattern q_2, if $q_1[i] \prec q_2[i]$ for $q_1[j] \simeq q_2[j]$ and $1 \leq j < i$. We say q_1 and q_2 are equally ranked, denoted by $q_1 \simeq q_2$, if $q_1[i] \simeq q_2[i]$ for $i = 1, 2, 3$.

The following lemma demonstrates the equivalence between triple patterns v and $\phi(v)$ for any containment mapping ϕ.

LEMMA 2. *If pattern V contains pattern Q under containment mapping ϕ, for any $v \in V$, $v \simeq \phi(v) \in Q$.*

PROOF. Because $V \sqsupseteq Q$ under containment mapping ϕ, for any $v \in V$, there is a triple pattern $q \in Q$ such that $\phi(v) = q$. Moreover, the triple pattern q is obtained by replacing variables appearing in v with variables or references according to ϕ. Therefore, for $l = 1, 2, 3$, $v[l] \simeq q[l]$ according to the definition of containment order where a variable is ranked equal with anything. The lemma is proved. □

4.2 Sortable View

While containment order connects v with $\phi(v)$ under any containment mapping ϕ, it is not sufficient to derive containment relationship. In particular, the order relationship between tripe patterns v_i and v_j at the view side cannot carry over to those $\phi(v_i)$ and $\phi(v_j)$ at the query pattern side even if $\phi(V) = Q$, i.e., if $v_i \prec v_j$ and $v \simeq \phi(v)$ for any triple pattern v, it can not be obtained that $\phi(v_i) \prec \phi(v_j)$. If not, the order can be used to sort triple patterns in view and query pattern, respectively. The containment mapping must relate the triple patterns at the same positions. Nevertheless, it can be proved that the sortable view can make the order relationship extendable from the view to the query. Theorem 2 claims the containment order can be used to filter out unpromising searching of containment mapping.

DEFINITION 6. ***Sortable View***
View $V = \{v_1, \cdots, v_l\}$ is called sortable if there exist an order such that $v_j \prec v_k$ for any $j < k$.

Given sortable view V, its triple patterns can be sorted increasingly in terms of containment order. Hence, let $V[i]$ be the ith triple pattern in its sorting. Given general pattern Q, its triple patterns can be arbitrarily serialized into an array and let $Q[i]$ be the triple pattern at the ith position under this serialization. However, among various serializations of Q, one is special, called V-serialization against sortable view V.

DEFINITION 7. ***V-Serialization***
If the serialization of Q satisfies $Q[i] \simeq V[i]$ for $i = 1..|V|$, this serialization is called V-serialization of Q.

THEOREM 2. *Given pattern Q against sortable view V, $V \sqsupseteq Q$ if and only if there is a containment mapping ϕ such that $\phi(V[i]) = Q[i]$ for V-serialization of Q.*

REMARK 1. *View V_2 is sortable and $V_2 \sqsupseteq Q_2$. However, Q_2 is not sortable. There are two different orders available for Q_2, as is shown below.*

$(Frank, coAuthor, ?o') \prec (?n', isMother, ?o') \prec (Bob, Marry, ?n')$

or

$(?n', isMother, ?o') \prec (Bob, Marry, ?n') \prec (Frank, coAuthor, ?o')$

While Q_2 is not sortable, its V_2-serialization is determined in the following.

$V_2[1]$	$V_2[2]$	$V_2[3]$
$(?p, coAuthor, ?o)$	$(?n, isMother, ?o)$	$(?m, Marry, ?n)$
$Q_2[1]$	$Q_2[2]$	$Q_2[3]$
$(Frank, coAuthor, ?o')$	$(?n', isMother, ?o')$	$(Bob, Marry, ?n')$

By Theorem 2, if a containment mapping ϕ exists, then $\phi(V_2[1]) = Q_2[1]$, $\phi(V_2[2]) = Q_2[2]$ and $\phi(V_2[3]) = Q_2[3]$.

REMARK 2. *Given pattern Q against sortable view V, its V-serialization can be efficiently obtained if it exists. Each triple pattern $q \in Q$ is put into the ith position of $Q[i]$ if $q \simeq V[i]$.*

Before giving the proof of Theorem 2, two basic lemmas are first proved which are extensions from Lemma 2 in the case of sortable views. Lemma 3 states that there is no more than one triple pattern in Q which is equivalent with $V[i]$. Lemma 4 further shows that containment mapping must relate equivalent triple patterns. These two lemmas together establish the foundation of Theorem 2 on the contribution of the equivalence relationship between $Q[i]$ and $V[i]$ to the containment mapping.

REMARK 3. *Based on Lemma 3 and 4, Theorem 2 claims that view containment can be reduced to comparison of triple patterns with respect to containment order.*

To simply the presentation, let $P(V, q)$ be the set of equivalent triple patterns of q, i.e., $P(V, q) = \{v \in V | v \simeq q\}$.

LEMMA 3. *Given sortable view V and pattern Q with the same length, if $V \sqsupseteq Q$ then for any $q \in Q$, $|P(V, q)| = 1$*

PROOF. Since $V \sqsupseteq Q$, there is a containment mapping ϕ such that $\forall q \in Q$, $\exists v \in V$, $\phi(v) = q$. By Lemma 2, $v \simeq q$, so $v \in P(V, q)$. If $|P(V, q)| > 1$, there is another triple pattern, say v', in V such that $v' \simeq q$. Since V is a sorted pattern, without loss of generality, we assume $v \prec v'$. Since $v \prec v'$, there exists l such that both $v[l]$ and $v'[l]$ are references and $v[l] \prec v'[l]$. However, we know that $v \simeq q$ and $v' \simeq q$, so $q[l]$ is a variable. This is a contradiction since a containment mapping has $\mathsf{var}(V)$ as its domain and $\mathsf{dom}(Q)$ as its range. The lemma is proved. □

LEMMA 4. *Given sortable view V and pattern Q with the same length, $V \sqsupseteq Q$ if and only if there is a containment mapping ϕ such that for any $q \in Q$, $\phi(P(V,q)) = \{q\}$.*

PROOF. *IF* part: If there is a containment mapping ϕ such that $\phi(P(V,q)) = \{q\}$, we have $|P(V,q)| = 1$ since V is a sorted pattern. Furthermore, if there is another triple pattern $q' \in Q$ such that $P(V,q') = P(V,q)$, then $\phi(P(V,q')) = \phi(P(V,q)) = \{q\}$. However, we have $\phi(P(V,q')) = \{q'\}$, this is a contradiction. Therefore, for each $v \in V$, we cannot find q and q' in Q such that $\phi(v) = q$ and $\phi(v) = q'$. Since $|V| = |Q|$, we must have ϕ is a bijection from V to Q. Therefore, we have $\bigcup_{q \in Q} P(V,q) = V$, this infers that $\phi(V) = Q$. That is, $V \sqsupseteq Q$.

ONLY IF part: Based on Lemma 2, if $\phi(v) = q$, then $v \in P(V,q)$. Based on our previously proved Lemma 3, $P(V,q) = \{v\}$ and therefore $\phi(P(V,q)) = \phi(\{v\}) = \{q\}$. □

Proof of Theorem 2. The if part is immediately obtained. Only if part: Since $V \sqsupseteq Q$, Q is V-serializable, i.e., $V[i] \simeq Q[i]$. At the same time, there is a containment mapping ϕ from V to Q. Due to Lemma 2, $\phi(V[i]) \simeq V[i]$. Combined with Lemma 4, we have $\phi(V[i]) = Q[i]$. □

4.3 Containment Searching Algorithm

Theorem 2 immediately results in Algorithm 1 for view containment. It receives pattern Q against sortable patterns V with the same length and returns the containment mapping ϕ if $V \sqsupseteq Q$, otherwise an empty mapping. After sorting the view V and serializing pattern Q, for each i, contain builds the containment mapping mapping ϕ_i from $V[i]$ to $Q[i]$ such that $\phi_i(V[i]) = Q[i]$. Finally, if $\phi = \bigcup \phi_i$ is a containment mapping then $\phi(V) = Q$, otherwise, $\phi = \emptyset$.

An example is given for building mapping ϕ_i. Let $V[i] = (?x,$ isMother, $?z)$ and $Q[i] = (?a,$ isMother, $Bob)$, we have $\phi_i = \{?x \mapsto ?a,\ ?z \mapsto Bob\ \}$ such that $\phi_i(V[i]) = Q[i]$.

Algorithm 1 contain(V, Q)

{ V is sortable and $|V| = |Q|$ }
1: initiate containment mapping ϕ with empty;
2: **if** there is no V-serialization of Q **then**
3: return $\emptyset$;
4: **end if**
5: **for** $i = 1$ to $|V|$ **do**
6: build mapping ϕ_i such that $\phi_i(V[i]) = Q[i]$;
7: $\phi = \phi \cup \phi_i$;
8: **if** ϕ is not a mapping **then**
9: return $\emptyset$;
10: **end if**
11: **end for**
12: return ϕ;

The testing in line 8 keeps ϕ is a mapping, i.e., there are no conflicts of mapping a variable to different images or different variables to an image.

When $|Q| > |V|$, the verification of whether $V \sqsupseteq Q$ involves iterative calls of contain(V, Q') for each $Q' \subset Q$ with the length of $|V|$. This naive iteration is time prohibitive. In Algorithm 2, we present contain$^+(V,Q)$, removing the pre-condition of $|Q| = |V|$. Although the algorithm still takes exponential time in the worst case, the number of candidate $Q' \subseteq Q$ is greatly reduced by exploiting the containment order. In the following, we use an example to illustrate its efficiency.

For patterns $V_3 = \{v_1 = (?x, \text{Marry}, ?y),\ v_2 = (?x, \text{isMother}, ?z)\}$ and $Q_4 = \{q_1 = (?c, \text{Marry}, \text{Bob}),\ q_2 = (?c, \text{isMother}, ?d),\ q_3 = (?d, \text{isBrother}, ?e)\}$. In the similar way to the previous example, $P(V_1, q_1) = \{v_1\}$, $P(V_1, q_2) = \{v_2\}$, but $P(V_1, q_3) = \emptyset$, which implies that there is no triple pattern in V_1 which can be mapped to q_3. Therefore, q_3 is excluded from candidates and the only possible candidate is $Q'_4 = \{q_1, q_2\}$. It is only necessary to invoke contain(V_1, Q'_4) once for further verification.

Algorithm contain$^+(V,Q)$ consists of two phases. At the first phase, it computes $P(Q,V) = \bigcup_{v \in V} P(Q,v)$. It is immediately verified that $P(Q,V) \subseteq Q$ and $V \sqsupseteq Q$ if and only if $V \sqsupseteq P(Q,V)$.

At the second phase, it builds a partition on $P(Q,V)$. With respect to sortable view V, two triple patterns q and q' are conflict, denoted as $q \oslash q'$, if and only if $P(V,q) = P(V,q')$. By applying the equivalent relation $\oslash$ on $P(Q,V)$ and a partition of $P(Q,V)$ is obtained, i.e., $P(Q,V) = \bigcup_{k=1..|V|} Q_k$, where Q_k is an equivalence class. By retrieving one triple pattern from each Q_k, a query pattern Q' is composed, i.e., $Q' = \{q_k \in Q_k | k = 1..|V|\}$. It is obvious that no conflict triple patterns exist in Q'. Let Ω be the collect of all such composed patterns Q' and input it to contain for further verification. The details are presented in Algorithm 2.

Algorithm 2 contain$^+$(V, Q)

{ V is sortable and $|Q| > |V|$ }
1: **if** $|P(Q,V)| \geq |V|$ **then**
2: partition $P(Q,V)$ in terms of equivalence relation $\oslash$;
3: **else**
4: return $\emptyset$;
5: **end if**
6: **for** each composed pattern Q' **do**
7: **if** contain$(V, Q') \neq \emptyset$ **then**
8: output Q' and ϕ; /*ϕ returned by contain()*/;
9: **end if**
10: **end for**

4.4 Sortable View Selection

In this part of the section, we discuss the feasibility of sortable views in RDF model. To be precise, the question is on the existence of an efficient algorithm to construct sortable views. We are also interested in finding certain approach to transform/rebuild unsortable views to sortable ones. Positive answers are provided in the following for both of the questions above.

Given pattern Q, a directed graph order(Q), called *ordered graph* of Q, can be constructed, where order(Q) $=< V_Q, E_Q >$. Let $V_Q = Q$ and a directed edge (q_i, q_j) from q_i to q_j is included in E_Q if and only if $q_i \prec q_j$. Therefore, there is a topological sort for order(Q) if it is acyclic.

THEOREM 3. *Pattern V is sortable if and only if there is a unique topological sort of order(V).*

PROOF. If part: If the topological sort of order(V) does not exist, then order(V) must be cyclic, i.e., there is a cyclic path, $v_1 \prec \cdots \prec v_1$, implying V is not sortable based on Definition 6. Hence, if a given V can be sorted, there is a topological sort of order(V).

Only if part: We first prove the fact that if there are more than one topological sort of order(V), there are at least two triple patterns with no edges between them. To obtain the topological sort, we repeat retrieving and deleting the node with indegree 0 on the current order(V) step by step. If there are two topological sorts, there must be a step where there are two triple patterns whose indegrees are both zero, meaning we have two choices of retrieval and deletion. Further, there are no edges between them. Hence there are two triple patterns in V without $\prec$ or $\succ$ relation, meaning V

Algorithm 3 **rewrite**(Q, Γ)

1: **for** each $V_i \in \Gamma$ **do**
2: **if** $P(V_i, Q) = V_i$ and $\mathsf{contain}^+(V_i, Q)$ **then**
3: insert $\phi_i(V_i)$ into $\Gamma(Q)$;
4: **end if**
5: **end for**
6: output $\Gamma(Q)$;

can not be sorted. Finally, the unique topological sort $v_1 \prec v_2 \cdots$ of order(V) determines the increasing sorting of triple patterns in pattern V. □

Clearly, Theorem 3 provides a way of testing whether a view is sortable. Given view V, construct order(V) and do topological sort on order(V). During the topological sort, if there are two nodes with indegree zero, V is not sortable, otherwise, V is sortable.

On the other hand, Theorem 3 provides an efficient way of generating sortable views. In a similar way, for view V, do topological sort on order(V). At each step of this sort, if there is one node with indegree zero, delete the node and put it into V', otherwise, delete one of them. Finally, V' is sortable.

REMARK 4. *If there is a triple pattern in V' which does not share variables with other triple patterns, it is isolated and should be deleted from V' further.*

In practice, given view V, generate sortable view V' from V in the above way. Repeat it on $V - V'$ for another sortable view V''. Finally, a view is partitioned into the union of sortable views. Thereafter, query are evaluated step by step via RDF query engine, say RDF-3x [20]. At each step, a sortable view is evaluated. Finally, join all these results together to finish the input query. During this procedure, it can be decided whether to materialize the sortable view.

5. REWRITING AND OPTIMIZATION

While previous section focuses on the property of sortable view for containment mapping search and feasibility. In this section, we first provide the basic complete query rewriting algorithm using sortable views. We then present an inverted index structure for fast retrieval of candidate views in the evaluation of particular pattern query Q. We also discuss how to optimize the selection of view from large candidate pool, to remove redundant views for expected evaluation cost minimization.

Consider our running example in Figure 1, where $\Gamma = \{V_1, V_2\}$, $V_1 \sqsupseteq Q_1$ but $V_2 \not\sqsupseteq Q_1$ as we have shown. It is desirable not to invoke $\mathsf{contain}^+(V_2, Q_1)$ for further verification. The following lemma provides a necessary requirement to filter out unpromising views. Recall that $P(V, Q) = \bigcup_{q \in Q} P(V, q)$.

LEMMA 5. *If $V \sqsupseteq Q$, then $P(V, Q) = V$.*

PROOF. It is clear $V \supseteq P(V, Q)$. We show $P(V, Q) \supseteq V$. For any $v \in V$, there exists $q \in Q$ such that $\phi(v) = q$. By Lemma 2, $v \in P(V, Q)$. Hence $P(V, Q) \supseteq V$. □

Based on Lemma 5, we present a complete algorithm rewrite$(Q, \Gamma))$ for pattern rewriting (See Algorithm 3). Specifically, the algorithm retrieves those promising patterns V_i in view base such that $P(V_i, Q) = V_i$ and then invoke $\mathsf{contain}^+(V_i, Q)$ for further verification.

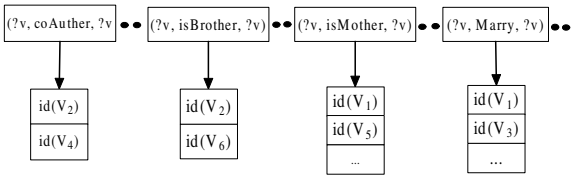


Figure 3: Example of inverted list index.

5.1 Inverted Index Structure

To support the retrieval of promising views, we build inverted lists and put all triple patterns in Γ into $\mathcal{V}$, i.e., $\mathcal{V} = \{(v, \mathsf{id}(V_i) | V_i \in \Gamma, v \in V_i\}$, where $\mathsf{id}(V_i)$ is the identification of V_i. Given $q \in Q$, let $P(\mathcal{V}, q) = \{(v, \mathsf{id}(V_i)) | q \simeq v, (v, \mathsf{id}(V_i)) \in \mathcal{V}\}$ and $P(\mathcal{V}, Q) = \bigcup_{q \in Q} P(\mathcal{V}, q)$. Finally, grouping $(v, \mathsf{id}(V_i))$ by $\mathsf{id}(V_i)$ component in $P(\mathcal{V}, Q)$, we can retrieve complete patterns V_i for further verification.

Unfortunately, the containment order is not desirable to compute $P(\mathcal{V}, q)$ since $\simeq$ is not a transitive relation. For example, $q_1 \simeq q$ and $q_2 \simeq q$ but no $\simeq$ between q_1 and q_2, meaning triple patterns ranked equal with q may not be grouped together. To tackle this problem, we define a total order, called *index order* for fast retrieval.

DEFINITION 8. ***Index Order on*** $\mathbb{D}$
Index order $(\preceq^i, \mathbb{D})$: 1) $\forall x, y \in \mathbb{D}_u$, $x \prec^i y$ iff x precedes y in terms of lexical order. 2) $\forall x \in \mathbb{D}_u, y \in \mathbb{D}_v$, $x \prec^i y$. 3) two variables $?x$, $?y \in \mathbb{D}_v$ are ranked equal, denoted by $?x \simeq^i ?y$.

DEFINITION 9. ***Index Order on*** $\mathbb{P}$
$(\preceq^i, \mathbb{P})$: A triple pattern $q_1 = (s_1, p_1, o_1)$ is ordered before triple pattern $q_2 = (s_2, p_2, o_2)$, if 1) $s_1 \prec^i s_2$, or 2) $s_1 \simeq^i s_2$ and $p_1 \prec^i p_2$, or 3) $s_1 \simeq^i s_2$, $p_1 \simeq^i p_2$ and $o_1 \prec^i o_2$. We say q_1 and q_2 are equally ranked, denoted by $q_1 \simeq^i q_2$, if $s_1 \simeq^i s_2$, $p_1 \simeq^i p_2$ and $o_1 \simeq^i o_2$.

LEMMA 6. *Index order $\preceq^i$ on $\mathbb{P}$ is a total order. Specifically, $\simeq^i$ is an equivalence relation on $\mathbb{P}$.*

PROOF. We only show the transitivity of $\preceq^i$, i.e., if $q_1 \preceq^i q_2$ and $q_2 \preceq^i q_3$, then $q_1 \preceq^i q_3$. Since $q_1 \preceq^i q_2$, $\exists l$, $q_1[l] \preceq^i q_2[l]$ for $l = 1, 2, 3$. Similarly, $\exists l'$, $q_2[l'] \preceq^i q_3[l']$ for $l' = 1, 2, 3$. W.l.o.g, we assume $l \leq l'$, we have $q_1[i] \simeq^i q_2[i] \simeq^i q_3[i]$ for $i < l$ while $q_1[l] \prec q_2[l] \simeq^i q_3[l]$. Hence, $q_1[l] \prec q_3[l]$ and therefore, $q_1 \preceq^i q_3$. Similarly, we can prove that $\simeq^i$ is an equivalent relation. □

Given an index order, the inverted list index is constructed by grouping the triple patterns with equivalence relationship under index order $\simeq^i$. Each inverted list keeps a list of the identifications $\mathsf{id}(V_i)$ of patterns V_i in Γ, such that V_i contains a triple pattern in this equivalence class. In Figure 3, we present an example of the inverted lists. Based on the definition, each of $\mathsf{id}(V_1)$ and $\mathsf{id}(V_2)$ is stored in two inverted lists for the two triple patterns in it. Note that all variables are replaced with a common variable symbol $?v$ in the index, since all variables are regarded as equal in the index order $\simeq^i$.

While the index order delivers complete equivalence classes in the triple space $\mathbb{P}$, this order does not directly guarantee the correctness when retrieving candidate patterns for a query Q, since $\simeq$ is different from $\simeq^i$. To retrieve a complete candidate set for $P(V_i, Q)$, it is important and necessary to fill the semantics gap between $\simeq$ and $\simeq^i$, i.e., by expanding the triples in the query Q before the candidate retrieval.

Generally speaking, the expansion is done by 1) replacing all variables with the unique variable symbol $?v$, and 2) generating

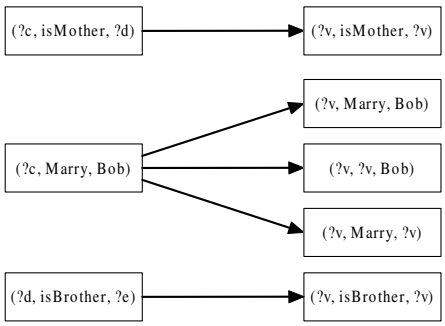


Figure 4: Example of query expansion on Q_2.

new triple patterns by generalizing references to variables, when there are more than one reference in the triple pattern. For triple pattern $q_2 = (?c, \text{Marry}, \text{Bob})$, for example in Figure 4, there are two references, Marry and Bob. Therefore, we expands on both references, substituting each reference with the common variable $?v$. This leads to three triple patterns to query on the inverted list index. The expansion operation is important, because it ensures that there is no promising patterns lost in the candidate retrieval. For triple pattern q, let $\mathsf{exp}(q)$ be the set of all expanded triple patterns via the above expansion process. Let $q_i \sqsupseteq q_j$ denote $\{q_i\} \sqsupseteq \{q_j\}$. The relation between $\simeq$ and $\simeq^i$ is summarized as follows.

LEMMA 7. *Given two triple patterns $q_i \sqsupseteq q_j$, $q_i \simeq q_j$ if and only if $\exists q \in \mathsf{exp}(q_j)$, $q \simeq^i q_i$.*

PROOF. We first give the following two observations, which are immediately verified based on the related definitions. First, given two triple patterns $q_1 \simeq^i q_2$, if triple pattern $q \simeq q_1$, then $q \simeq q_2$. Second, for $q' \in \mathsf{exp}(q)$, $q' \simeq q$.

If part: Since $q \in \mathsf{exp}(q_j)$, $q_i \simeq q$ based on the second observation. Further, because $q \simeq^i q_j$, we have $q_i \simeq q_j$ based on the first observation.

Only if part: Since $\{q_i\} \sqsupseteq \{q_j\}$ and $q_i \simeq q_j$, we can obtain the containment mapping ϕ such that $\phi(q_i) = q_j$. Hence, $\phi^-(q_j) \simeq^i q_i$, where ϕ^- is the inverse mapping of ϕ. It is clear $\phi^-(q_j) \in \mathsf{exp}(q_i)$. This finishes the proof. □

Since every triple pattern has at most two variables, there are at most three triple patterns after the expansion, the expanded triple patterns do not introduce heavy additional workloads. For each expanded triple pattern, the system quickly marks the equivalent inverted lists w.r.t $\simeq^i$. Finally, the algorithm counts the number $\#\mathsf{id}(V_i)$ of $\mathsf{id}(V_i)$ appearing in the marked lists. Note that the appearance of an identification in a list is counted once. For $\#\mathsf{id}(V_i)$, we have the following theorem.

THEOREM 4. *Given sortable view in base Γ, for any V_i in Γ, $P(V_i, Q) = V_i$ if and only if $\#\mathsf{id}(V_i) = |V_i|$.*

PROOF. Note that identification $\mathsf{id}(V_i)$ for pattern V_i can appear at most once in any list because any pattern in the view base is kept sortable. If part: since $\#\mathsf{id}(V_i) = |V_i|$, each list containing $\mathsf{id}(V_i)$ is visited, meaning $p(V_i, Q) = V_i$. Only if part: since $P(V_i, Q) = V_i$, for each $q \in Q$, all inverted lists whose entries are ranked equal to q in terms of $\simeq^i$ are marked. Remember $\#\mathsf{id}(V_i)$ is the number of appearances of $\mathsf{id}(V_i)$ in the marked lists. Since each $\mathsf{id}(V_i)$ can appear at most once in any list, $\#\mathsf{id}(V_i) = |V_i|$. □

The details of pattern rewriting method with index are summarized in Algorithm $\mathsf{rewrite}^+(Q, \Gamma)$ (see Algorithm 4). The algorithm receives pattern Q and the set Γ of views and returns the rewriting $\Gamma(Q)$ of Q. It consists of two phases. At the first phase, it finds all promising patterns in the view base based on Theorem 4 (line 6). At the second phase, for each promising pattern, it invokes $\mathsf{contain}^+(V_i, Q)$ to do further verification.

Algorithm 4 $\mathsf{rewrite}^+(Q, \Gamma)$

```
1: for each q ∈ Q do
2:     expand q;
3:     mark the entry equal to one of the expansions;
4: end for
5: count #id(V_i) among the marked lists;
6: for each #id(V_i) = |V_i| do
7:     invoke contain+(V_i, Q) for further verification;
8:     insert φ_i(V_i) into Γ(Q) if V_i ⊒ Q;
9: end for
```

REMARK 5. *The process of marking inverted lists can be improved by applying a B^+-tree index structure on the entries of the inverted lists due to the total order $\preceq^i$.*

5.2 Expected Cost Optimization

Theorem 1 states a possible way of exploiting views to evaluate pattern matching. However, the evaluation of $\Gamma(Q)$ defined in Definition 3 possibly uses redundant views. To be precise, it is possible that $\phi_k(V_k) \subseteq \phi_i(V_i) \cup \phi_j(V_j)$ or $\phi_k(V_k) \supseteq \phi_i(V_i) \cup \phi_j(V_j)$. The term $[\phi_k(V_k)]$ in the former case or $[\phi_i(V_i)] \bowtie [\phi_j(V_j)]$ in the latter case can be deleted from the expression of $[\Gamma(Q)]$ in Definition 3 without compromising the correctness of Theorem 1. Such views are called redundant views in pattern rewriting. Formally, if there is view base $\Gamma' \subset \Gamma$ such that $[\Gamma'(Q)] = [\Gamma(Q)]$ for any graph G, The views in $\Gamma - \Gamma'$ are called redundant views.

Redundant views are pruned from pattern rewriting such that the cost of pattern rewriting is minimized. To be precise, each view V_i is associated with a cost $c(V_i)$ which indicates the cost of obtaining $[\phi_i(V_i)]$ from $[V_i]$. The cost $c(\Gamma(Q))$ of pattern rewriting $\Gamma(Q)$ is the sum of $c(V_i)$ for $\phi_i(V_i) \in \Gamma(Q)$. That is,

$$c(\Gamma(Q)) = \sum_{\phi_i(V_i) \in \Gamma(Q)} c(V_i)$$

Therefore, it becomes an optimization problem to find $\Gamma' \subseteq \Gamma$ such that the cost of $c(\Gamma'(Q))$ is minimized under the constraint of $[\Gamma'(Q)] = [\Gamma(Q)]$.

DEFINITION 10. ***Optimum Pattern Rewriting***
The pattern rewriting $\Gamma'(Q)$ is optimum if its cost $c(\Gamma'(Q))$ is minimized under the constraints of $\Gamma' \subseteq \Gamma$ and $[\Gamma'(Q)] = [\Gamma(Q)]$ for any RDF graph.

The optimization problem we consider here can be reduced to set cover problem [18]: $\bigcup_{\phi_i(V_i) \in \Gamma(Q)} \phi_i(V_i)$ acts as the universe set and $\Gamma(Q)$ as the collect of sets to cover the universe set such that the union of the them includes it with the minimized cost. It worths noting that when $c(V_i) = 1$, this reduction can output the minimized number of views for pattern rewriting.

REMARK 6. *When put $Q - \Gamma'(Q)$ into Γ for any $\Gamma' \subset \Gamma$, the optimum pattern rewriting may ignore a view V_i in its rewriting even if $V_i \sqsupseteq Q$. That is, some views are not always useful when the cost of using them exceeds the cost of ignoring them.*

para.	description	default
$\mathcal{L}$	average length of patterns	10
$\mathcal{V}$	average number of variables	5
M	mode of triple generation	D
V	mode of variable generation	D

Table 3: Parameter settings in experiments

6. EMPIRICAL EVALUATION

Based on Theorem 1, we implement a view aware evaluation engine,sSeg, which segments a query into parts. Some parts can be answered by materialized views. The final answer is obtained by joining answers to these parts. In particular, it first rewrites pattern Q into $\Gamma(Q)$ via rewrite(Q, Γ) and then evaluate Q via Theorem 1. For comparison, we use Rdf-3x [20], a RISC-style engine for RDF queries as the baseline, as well as our backend RDF query engine for obtaining $[Q - \Gamma(Q)]$ component in Theorem 1.

For testing the efficiency of rewrite(Q, Γ), we also implement another view aware evaluation engine, gSeg. It is a little different from sSeg by employing graph containment searching [7] for pattern rewriting, reducing containment mapping to subgraph isomorphism.

We tested all the approaches on the data sets used in [20]. Since most of the data sets display similar results, we only report our experimental results on *Yago* data set in this paper. The Yago data set contains 40,114,899 distinct triples and 33,951,636 distinct strings, consuming 3.1 GB as (factorized) triple dump. Rdf-3x [20] needs 2.7 GB for all indexes and the string dictionary. All the programs, for both sSeg and gSeg, are compiled with GCC 4.3 in Ubuntu 10.04 LTS. The codes of Rdf-3x [20] are obtained from open source. All the experiments were run on a server with Intel(R) Core(TM)2 CPU 4300@1.80GHz and 2G main memory.

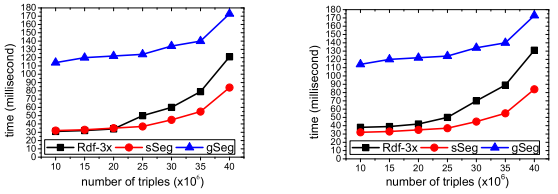
Pattern Generation. We used random walk on Yago data set to generate query patterns. First of all, an edge exists between two triples, if they share at least one common reference, on subject, predict or object. For a particular triple p, the set $N(p)$ contains all its neighboring triples. Given a subgraph (a set of triples) $P \subseteq G$, the neighborhood of P is defined as $N(P) = \bigcup_{p \in P} N(p)$. The degree of a triple p is $|N(p)|$. The frequency of a reference is the number of triples in the data set which contain the reference.

There are several parameters controlling the generation of the query patterns, including *average pattern length* $\mathcal{L}$, *variable number* $\mathcal{V}$, *triple mode* M and *variable mode* V. While the first two parameters are straightforward, the last two parameters are much more complicated, which are closely related to the generation mechanism. Every pattern Q_i was generated through some random walk on the data graph. Before the generation, the generator randomly selected the number of triple patterns in Q_i, following the uniform distribution on $[0.5L, 1.5L]$. After the selection of the triple number n, a two-phase generation scheme was invoked. In the first phrase, the generator randomly picked up a triple from the data as the seed, i.e $Q_i = \{t_1\}$. It then continued adding neighbors to the current triples in Q_i, until there are exactly n triples in Q_i. For neighbor set $N(Q_i)$, there are different strategies on the selection of next triple. If M =U, all the neighbors were chosen uniformly. If triple mode M =F (M =D resp.), a triple in $p = N(Q_i)$ was selected with probability proportional to the appearance frequency of p (the degree of p, i.e. $N(p)$, resp.). In the second phase, some of the references were selected and replaced by the variables. Similar to the first phase, the variable mode V has three options, including U, F and D, which follows uniform selection, frequency-based selection and degree-based selection respectively.

View Selection. We first generated $10,000$ patterns and then applied algorithms for frequent pattern discovery [24] to mine frequent patterns frequently included in query patterns. The mined frequent patterns were inserted into view base Γ_g and their answers were materialized in relational tables. To run sSeg, we keep all views in view base Γ_s sortable. We deleted the minimum number of triple patterns from a non-sortable pattern such that the left triple patterns form a sortable one (reference Theorem 3).Views with one triple pattern are deleted from Γ_s. If there is a triple pattern which shares no common variables with others in a view, this view is also deleted from Γ_s. In this way, view base Γ_g was transformed to Γ_s.

To reduce the effect of randomness, all of our experiments were run on 100 different queries. The average of these results are reported in this section. To test the impact of the correlation between the query patterns and patterns in view base, we also controlled the coverage $C_{\Gamma_g}(Q)$ of Γ_g, i.e. $C_{\Gamma_g}(Q) = |Q_{\Gamma_g}|/|Q|$. In the same way, $C_{\Gamma_s}(Q) = |Q_{\Gamma_s}|/|Q|$.

Query Efficiency. Figure 5 shows the query processing time when varying the cardinality of the database size from 10×10^6 to 40×10^6. In particular, the tests conducted on two query workloads generated with two different parameter settings. The results are presented in Figure 5(a) and 5(b), respectively. On both query workloads, the average query processing time of all methods increase exponentially with the expansion of the triple database. It can be observed that the overall query efficiency of gSeg is significantly worse than the standard RDF data engine Rdf-3x. On the other hand, sSeg presents competitive performance against Rdf-3x. When the number of triples reaches 40×10^6, the average processing time of sSeg is faster than Rdf-3x by 25%, on both of the workloads. This validates the effectiveness of sSeg especially for large data set.



(a) $L = 11$, M=D and V=D (b) $L = 11$, M=F and V=F

Figure 5: Scalability tests on two query workloads

In Figure 6, we discuss the impact of the coverage C_{Γ_g} on query processing efficiency. By varying C_{Γ_g} from 10% to 50%, both sSeg and gSeg reduce the running time when more patterns in the view base are expected to cover the coming queries. Rdf-3x, on the other hand, is unaffected, because it does not rely on the view base for query processing. The significant difference between the two view-based approaches arises from the expensive containment decision employed in gSeg. Again, the overhead for sSeg looks ignorable.

In Figure 7, we discuss the relationship between answer cardinality and query processing efficiency. Specifically, we generate 10 groups of random queries and record the number of matches on the triple patterns from the graph database G. Intuitively, the query processing takes more CPU time when there are more matchings with the queries. However, we did observe some exceptions in some of the experiments, as is shown in the figure. In particular, both gSeg and Rdf-3x show large variances on the computation time. For example, gSeg and Rdf-3x take more than 2.2 seconds when there are 4120 matchings. Our solution sSeg, on the other hand, shows more stable performance. sSeg finishes the

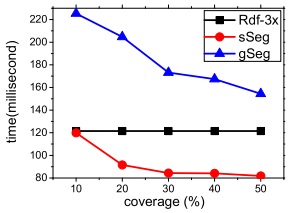


Figure 6: Coverage vs. time

whole computation within 1 second, even when there are more than 15,000 matchings.

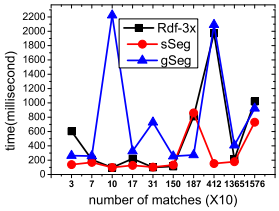


Figure 7: Number of matches vs. time

In Figure 8, we evaluate the influence of the query length, i.e. the average number of the triple patterns in the queries. The results in the figure show that sSeg is more scalable than the other two, in terms of the query length. The CPU time of gSeg increases sharply when the length is lifted from 11 to 13, while the computation time of sSeg remains within linear increase. The abrupt increase of computation cost of gSeg roots in its exponential complexity.

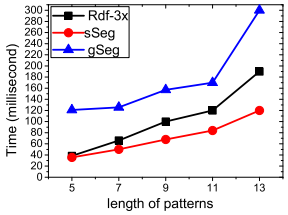


Figure 8: Query length vs. time

Finally, in Figure 9, we compare the methods when the number of variables in the queries expanded from 3 to 11. The number of variables is increased by replacing some of the same variables in triple patterns with different variables. For example, the variable $?x$ appears n times and some of them are replaced with new ones. The time cost increases as the number of unique variables increases in all methods. We also observe that the number of matchings increases with the increasing number of unique variables since more variables lead to less joins between triple patterns and therefore more matchings.

In our experiment, we find Rdf-3x can lead to machine crash when it evaluates some query patterns of more than 9 triple patterns. When the machine crashes, we delete the query pattern it is evaluating and continue with the remaining queries. The effect of the failure is not recorded.

Summary. Except for the length of query patterns, the number of matchings, as well as the intermediate partial matchings, makes

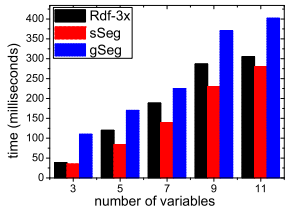


Figure 9: Number of variables vs. time

a significant impact on query efficiency. View-based approaches, especially sSeg, exhibit a good scalability in terms of the size of database, as well as pattern length.

Pattern Rewriting. In this part, we focus on the evaluation of pattern rewriting involved in sSeg and gSeg with respect to the average length of patterns/views and the size of view base. We report the results on CPU time. We also report the relation between Γ_g and Γ_s to verify the reasonableness of sortable views.

In Table 4, we report the impact of average length $\mathcal{L}$ of patterns in view base Γ_g . A set of 100 patterns with 10 triple patters each was generated as query patterns Q. When $\mathcal{L}$ on Γ_g was varied from 3 to 9, the difference between $\mathcal{L}$ on Γ_g and Γ_s was ignorable. This means the restriction of sortable views was applicable. It is more interesting that view base with short patterns can improve the coverage as shown in C_{Γ_s} column. As expected, the time cost for gSeg is much higher than sSeg because the length of $Q - \Gamma(Q)$ is increased as keeping $C_{\Gamma_g} = 30\%$. sSeg shows stable time cost with minor increase. The fluctuation is due to the structure updates of view base when the average length $\mathcal{L}$ of view base is enlarged. Despite of the fluctuation, our serialization method is a clear winner.

$\mathcal{L}$ on Γ_g	$\mathcal{L}$ on Γ_s	C_{Γ_s}	sSeg	gSeg
3	3	30%	1.1	110.6
5	4	33%	2.1	121
7	6	37%	3.4	204
9	7.5	41%	3.6	417

Table 4: Average length L of patterns in Γ_g vs. Time of pattern segmentation (milliseconds, $C_{\Gamma_g} = 30\%$)

In Table 5, we report the impact of the average length of query patterns on a given view base with default $\mathcal{L}$. The coverage fluctuates around the default coverage 30% in both methods. The minor advantage of sSeg stems from the short average length of patterns in view base Γ_s. However, with respect to time efficiency, sSeg is a clear winner again with enlarged advantages in the case of long query patterns.

L	C_{Γ_g}	C_{Γ_s}	sSeg	gSeg
9	30%	32%	1.1	110.6
11	33%	34%	2.1	121
13	31%	31%	3.4	204
15	32%	37%	3.6	417

Table 5: Average length L of query patterns vs. Time cost of pattern segmentation (milliseconds, $C_{\Gamma_g} = 30\%$) for patterns with average length 9

In Table 6, we list the average time of pattern rewriting of both sSeg and gSeg, on view bases with different cardinalities. The results in the table shows that our sSeg method is faster than gSeg

by almost two orders of magnitude. The efficiency comparisons verify our statement on the superiority of our serialization-based pattern rewriting in terms of efficiency. With the expansion of view base, the pattern rewriting time of both sSeg and gSeg increase. The exponential increase of time cost in gSeg is observable while the minor increase in sSeg. This phenomenon verifies the exponential time cost of sub graph isomorphism decision underlying gSeg. The efficiency advantage of sSeg stems from the restriction of sortable views.

$R_g = \|\Gamma_g\|$	sSeg	gSeg
1,000	1.1	172.6
2,000	2.7	335.1
3,000	3.4	456.2
4,000	3.6	603.5
5,000	4.3	800.0

Table 6: Impact of $|\Gamma_g|$ on time cost (milliseconds)

Effectiveness of Enhancement. To test the effectiveness of some enhancements in algorithm $\mathsf{contain}^+(V, Q)$, we implemented a comparison version $\mathsf{contain}^-(V, Q)$, where we naively enumerated all sub set Q' of Q such that $|Q'| = |V|$. Then, for each Q', we invoked $\mathsf{contain}(V, Q')$ for further verification.

We fixed the length of pattern V with 4 and slide the length of Q from 4, 6, 8 to 10. The time cost ratio of $\mathsf{contain}^-(V, Q)$ to $\mathsf{contain}^+(V, Q)$ is listed in Figure 10. The ratio exhibits an exponential increase with increasing length of pattern, verifying the effectiveness of our enhancements embedded in $\mathsf{contain}^+(V, Q)$. In particular, the advantage of $\mathsf{contain}^+(V, Q)$ comes from the serialization-enabled reduce of search space.

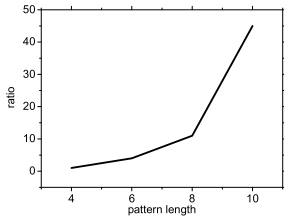


Figure 10: Time Cost Ratio of $\mathsf{contain}^-$ to $\mathsf{contain}^+$

7. CONCLUSION AND FUTURE WORK

In this work, we put the ends of answering queries using views together, exploiting sortable view selection at one end on behalf of pattern rewriting at the other end. A compact index for pattern rewriting and optimization for pruning redundant views are further considered. The integration of our pattern rewriting and existing optimizers will be detailed in our future work.

8. ACKNOWLEDGMENTS

Chong, Chen, Shu and Qi are supported by the National Natural Science Foundation of China under grant No. 60973023. Zhang is partly supported by Singapore A*STAR's Human Sixth Sense Project (HSSP) in Advanced Digital Sciences Center (ADSC).

9. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB Journal*, 18(2), 2009.

[2] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *Proc. of PODS '98*.

[3] M. Arenas, C. Gutierrez, and J. Pérez. Foundations of rdf databases. 2009.

[4] R. Castillo and U. Leser. Selecting materialized views for rdf data. In *Proc. of ICWE'10*, 2010.

[5] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. of STOC '77*.

[6] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proc. of ICDE '95*.

[7] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards graph containment search and indexing. In *Proc. of VLDB '07*.

[8] D. Chen and C.-Y. Chan. Minimization of tree pattern queries with constraints. In *Proc. of SIGMOD'08*.

[9] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient sql-based rdf querying scheme. In *Proc. of VLDB '05*.

[10] S. Cohen, W. Nutt, and Y. Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM Trans. Database Syst.*, 31(2), 2006.

[11] F. T. L. De Virgilio, Roberto; Giunchiglia. *Semantic Web Information Management*. Springer Heildelberg Dordrecht London New York, 2010.

[12] V. Dritsou, P. Constantopoulos, A. Deligiannakis, and Y. Kotidis. Optimizing query shortcuts in rdf databases. In *Proc. of ESWC'11*, 2011.

[13] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *Proc. of PODS '97*.

[14] G. H. Fletcher and P. W. Beck. Scalable indexing of rdf graphs for efficient join processing. In *Proc. of CIKM '09*.

[15] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View selection in semantic web databases. *PVLDB*, 5(2), 2011.

[16] P. Godfrey, J. Gryz, A. Hoppe, W. Ma, and C. Zuzarte. Query rewrites with views for xml in db2. In *Proc. of ICDE '09*.

[17] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4), 2001.

[18] E. t. Jon Kleinberg. *Algorithm design*. Person Education, Inc., 2006.

[19] C. Li. Computing complete answers to queries in the presence of limited access patterns. *The VLDB Journal*, 12(3), 2003.

[20] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1), 2010.

[21] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1), 2008.

[22] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proc. of WWW '08*.

[23] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1), 1976.

[24] J. X. Yu, Z. Chong, H. Lu, and A. Zhou. False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In *VLDB'04*, 2004.

[25] J. X. Yu, X. Yao, C.-H. Choi, and G. Gou. Materialized view selection as constrained evolutionary optimization. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 2003.