Front. Comput. Sci.
DOI 10.1007/s11704-012-1208-x

RESEARCH ARTICLE

Evaluation of RDF queries via equivalence

Weiwei NI[1], Zhihong CHONG (✉)[1], Hu SHU[1], Jiajia BAO[1], Aoyin ZHOU[2]

1 School of Computer Science and Engineering, Southeast University, Nanjing 210096, China
2 Software School, East China Normal University, Shanghai 200062, China

Abstract Performance and scalability are two issues that are becoming increasingly pressing as the resource description framework (RDF) data model is applied to real-world applications. Because neither vertical nor flat structures of RDF storage can handle frequent schema updates and meanwhile avoid possible long-chain joins, there is no clear winner between the two typical structures. In this paper, we propose an alternative open user schema. The open user schema consists of flat tables automatically extracted from RDF query streams. A query is divided into two parts and conquered on the flat tables in the open user schema and on the vertical table stored in a backend storage. At the core of this divide and conquer architecture with open user schema, an efficient isomorphic decision algorithm is introduced to guide a query to related flat tables in the open user schema. Our proposal in essence departs from existing methods in that it can accommodate schema updates without possible long-chain joins. We implement our approach and provide empirical evaluations to demonstrate both the efficiency and effectiveness of our approach in evaluating complex RDF queries.

Keywords divide and conquer architecture, open user schema, RDF query streams

1 Introduction

One of the goals of the semantic web is to enable integrating and sharing data across different applications and organizations. In this sense, the semantic web can be viewed as a global database [1]. One difference between the semantic web community and the relational database community is in its choice of the resource description framework (RDF) data model. RDF is a schema-relaxable or schema-free data model, representing data as statements describing resources [2,3]. RDF techniques can be helpful to solve semantic heterogeneity problems faced by the database community over the years. Also, database techniques can be explored to improve the performance and scalability of RDF data queries which are now bottlenecks of the semantic Web vision.

Data in the RDF model can be directly stored in a relational table, a triple table [2,4,5], of three columns representing subject, property, and object. An identified resource with semantic meaning can be constructed by recursively joining the triple table. In this strategy, performance degradation due to overhead join or union operations trades off against the schema flexibility required in the open world of the world wide web.

Besides employing index or optimization techniques to speed up joins [5–7], the proliferation of expensive joins can be controlled by exploring schema information of RDF data. Resource description framework schema (RDFS) or ontology web language (OWL), envisioned as a schema of RDF data, can be used to encode the formal semantics of RDF data, which can be easily combined into RDF data storage and RDF data query optimization, where triples can be fixedly combined into a flat table based on RDFS or OWL [8,9]. The performance is achieved because the data can be fetched directly from flat tables without many joins or unions. However, the previously fixed flat tables can not accommodate the variety of or updates on a flexible RDF schema which is definitely necessary in the open world of the World Wide Web.

Received September 28, 2011; accepted April 21, 2012

E-mail: wni@seu.edu.cn

To conquer the dilemma of either vertical or flat tables, we proposed open user schema (OU-Schema [10])[1]. Semantically, an OU-Schema is a relational schema consisting of a set of flat tables. Operationally, an OU-Schema is continuously extracted from users' query patterns. For example, frequent query patterns are extracted by streaming algorithms [11,12] and transformed into flat tables. The automatically extracted schema, rather than prior fixed, is therefore open to schema updates.

Pattern match is the core of the current RDF standard query language, simple protocol and RDF query language (SPARQL), and a query pattern is the conjunctive of triples. For example, in the following table, the query pattern Q describes books whose authors win prizes.

Q =	{(?b,hasAuthor,?a),	(?a,wonPrize,?z) }
?b	?a	?z
Book 1	Smith	Nobel prize
Book 2	Alice	Honor prize
⋮	⋮	⋮

A variable prefixed with ? in Q which can be matched to identified resources in the World Wide Web. The matches of variables in Q can be expressed as a relation or a table. The query pattern Q acts as the relation name and variables as attribute names. A complex query pattern is divided into two parts correspondingly, one part evaluated on the OU-Schema without the proliferation of join operations and the other on the triple table with the spirit of the flexible RDF data model. Therefore, we present a divide and conquer architecture, referred to as DC-Arc, to solve the dilemma of either vertical table or flat table storage. What's more, DC-Arc with OU-Schema can be seamlessly integrated with existing query engines without much effort by configuring it in front of other query engines.

From the view of data streams [11,12], RDF queries are continuously issued into a query engine and considered as a stream of queries. The query patterns included in most queries can be extracted from query streams using algorithms for data streams, e.g., algorithms for frequent pattern discovery [11,12]. The matches of the extracted query patterns Q can be materialized in the tables of $Q(?x_1, ?x_2, \dots)$ and ready for incoming queries. Consequently, RDF storage is divided into two parts, materialized query pattern matches and a triple table of RDF triples at the backend storage, RDF-3x [4] for example. OU-Schema distinguishes itself from the schema of flat tables in that it is automatically extracted from query streams and therefore open to schema updates.

The key point of DC-Arc with OU-Schema is how to efficiently guide a query evaluation to related tables in an OU-Schema. A similar problem is studied in the property table method [8], optimization with materialized views in relational tables [13]. In property table [8], the schema information of RDFS/OWL or bound properties can be used to guide a query evaluation to related property tables. But in OU-Schema, no such information can be explored. Therefore, DC-Arc with OU-Schema faces two key problems: the efficiency problem of query evaluation guided by OU-Schema and the effectiveness problem of OU-Schema maintenance for incoming queries.

The solution to problems in DC-Arc with OU-Schema is an efficient isomorphic decision algorithm. With the isomorphic decision algorithm at hand, query patterns with the same semantics can be related although they are in different forms. Hence, a query is efficiently divided into pieces of subqueries which are semantically equal to query patterns in the OU-Schema. As a result, the query is guided to related flat tables in the OU-Schema and a triple table at the backend storage.

Our key contributions are summarized as follows:

1. The notion of the OU-Schema is proposed to solve the dilemma of either vertical or flat structures of RDF storage.
2. An efficient method for deciding equivalent query patterns is given to guide a query to related tables in an OU-Schema.
3. Methods for extracting common patterns semantically included in query streams are also presented so that the OU-Schema is open to schema updates.
4. Seamlessly integrated with existing engines, the prototype of DC-Arc with OU-Schema is implemented and experiments are performed to empirically evaluate our approach.

After the overview of related work in Section 2, and preliminaries presented in Section 3. DC-Arc with OU-Schema and its problems are formally defined in Section 4. Isomorphic decisions as the basic block to guide a query to related tables in OU-Schema is studied in Section 5 before the details of evaluating a query with the help of OU-Schema in Section 6. The effectiveness problem of OU-Schema maintenance is

[1] In [10] we presented an earlier, shorter version of this work, only the main steps of decomposing queries were outlined while in this long version, details are presented, optimizations are included, and proofs and analysis are extended.

discussed in Section 7. After empirical evaluation in Section 8, we conclude our work and give future work in Section 9.

2 Related work

In general, we store RDF data in either vertical structures [2], e.g., a general triple table [4,5,14], or flat structures [8,9], no matter whether employing native RDF storage directly built on file systems [15] or a database using relational or object relational databases [8,16].

In the general triple table of vertical structures, as well as two-column table for each property [2], three columns, representing subject, predicate and object components of RDF data, can maintain the essence of the schema-free RDF data model but incurs possible long-chain joins to obtain a flat structure [7,14]. However, data in flat tables, say in a property table [8], with more than three columns can avoid long-chain joins but mean that frequent updates of data schema are not possible. Therefore, various methods are explored to resolve the dilemma. As shown in Fig. 1 at the left side, from the view of RDF data, flat tables are designed based on prior fixed RDFS/OWL [8], or by materializing/pre-computing intermediate query results in flat tables [8,17,18], or by partitioning the RDF graph into sub-graphs [19] to admit schema update. Index structures [4,5]and optimization [7] techniques are also explored to speed up join operations.

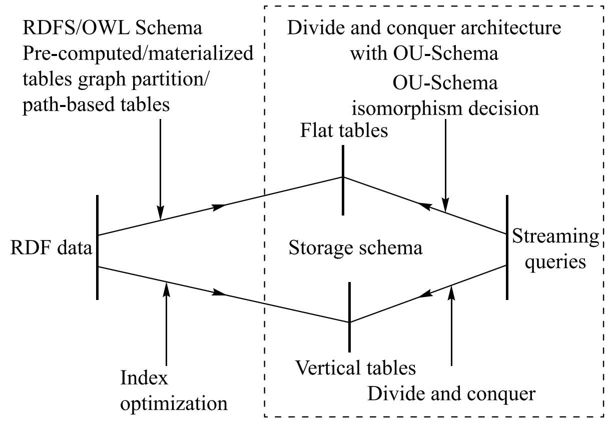

RDFS/OWL Schema
Pre-computed/materialized tables graph partition/ path-based tables
Divide and conquer architecture with OU-Schema
OU-Schema isomorphism decision
Flat tables
RDF data
Storage schema
Streaming queries
Vertical tables
Index optimization
Divide and conquer

Fig. 1 The state of the art and our proposal

However, most existing methods exploring either vertical or flat structures focus on RDF data structures while ignoring query structures. In Fig. 1, our work stands at the opposite end of existing work by considering structures of queries. The flat tables in an OU-Schema are extracted from streaming queries, rather than in advance fixed by RDF data. A query can be evaluated on both flat tables in an OU-Schema and vertical structures in a backend storage.

We share similar ideas of common expression [20], automating schema design [21]/self-tuning databases [22], optimizing queries with materialized views in relational databases [13], XML queries [23,24], and demand-driven caching in a multiuser environment [25]. In addition to guiding a query to related pieces of data in a large data set, query execution performance heavily depends on the materialized views in the underlying physical design. Our work distinguishes itself in the following aspects. Firstly, query optimization with materialized views in a relational database [13] is considered in the case of prior fixed relational data schema while the RDF data model is schema-relaxed/free. In the XML data model, the compact storage of materialized XML views is focused [23,24]. Secondly, The prior fixed schema is helpful for binding queries with materialized views when no schema information can be explored to guide a query pattern to related tables that are automatically extracted. Thirdly, besides enabling OU-Schema to be automatically configured over query streams, it is not trivial to guide a query to related tables in an OU-Schema as it involves an isomorphic decision between graphs as shown in Section 3.

It is worth noting that there are strong relations between OU-Schema and materialized views although they are different in essence. The main differences are: 1) in materialized views, queries are specified and then results are materialized after answers returned while in our cases, open schema is mined from many queries; 2) the focus of this paper is an index structure to exploit OU-Schema whereas materialized views are physical storage. That is, our index is used to access the physical storage.

3 Preliminaries

Following the semantics in [3], we provide an algebraic formalization of the core fragment of SPARQL over RDF. We do not consider blank nodes and RDFS vocabularies for simplicity as in [3].

The two sets U and V are pairwise disjoint infinite sets, where U represents the set of URIs and V the set of variables. Let triple space $\mathbb{G} = (U \times U \times U)$ and triple pattern space $\mathbb{P} = (U \cup V) \times (U \cup V) \times (U \cup V)$.

Definition 1 RDF triple: an RDF triple is a triple $(s, p, o) \in \mathbb{G}$, where s represents subject, p the predicate, and o the object.

Definition 2 RDF graph: an RDF graph G is a subset of $\mathbb{G}$, containing RDF triples.

Definition 3 RDF pattern/query pattern: a triple pattern is a triple $q \in \mathbb{P}$. The set Q of triple patterns is called an RDF pattern or a query pattern.

Let var(Q) denotes the set of variables occurring in a query pattern Q. $\mathbb{M}$ is the family of all one-to-one partial functions from V to U. For a map $\mu \in \mathbb{M}$, its domain is dom(μ). In this paper, a map μ can be expressed as a set of $\{?x \mapsto u | ?x \in V, u \in U\}$, where $?x$ in V is mapped to a URI in U. Hence, $?x \in \mathrm{dom}(\mu)$, $\mu(?x) = u$ iff $?x \mapsto u \in \mu$. Therefore, given an RDF pattern Q, if $\mathrm{dom}(\mu) = \mathrm{var}(Q)$, then $\mu(Q)$ is an RDF graph which is obtained by replacing each variable in var(Q) according to μ.

Definition 4 Pattern match: given a query pattern Q on RDF graph G, the match $[Q]_G$ of Q on G is the set $\{\mu(Q) | \mathrm{dom}(\mu) = \mathrm{var}(Q), \mu(Q) \subseteq G\}$.

In fact, $[Q]_G$ contains pieces of graph G which matches the query pattern Q.

4 OU-Schema and problem statement

Since pattern matching is the core of the current RDF query standard SPARQL, we focus our effort on improving the scalability of pattern matching which is currently a bottleneck of the RDF query process. As addressed in [6,26], a long query pattern match involves long-chain join operations and therefore its evaluation performance is greatly degrades. From the view of data streams [11,12], a triple pattern corresponds to an item and a query pattern to a set of items. Frequent query patterns over query streams correspond to common patterns included in many query patterns with high probability. Therefore, the matches of frequent query patterns can be materialized to reduce join operations if an input query pattern semantically includes the materialized frequent query patterns and therefore, the scalability of pattern matching can be improved.

For example, in Table 1, the query pattern $\widetilde{Q_i}$ and var($\widetilde{Q_i}$) forms a relational table $\widetilde{Q_i}(\mathrm{var}(\widetilde{Q_i}))$, where $\widetilde{Q_i}$ acts as the relation name, each variable in var($\widetilde{Q_i}$) plays the role of an attribute name in the relational table. And, $[\widetilde{Q_i}]$ is the instance of the schema $\widetilde{Q_i}$.

Definition 5 OU-Schema: an OU-Schema is a relational schema which consists of a set Γ of query patterns $\widetilde{Q_i}$.

In the following, a query pattern $\widetilde{Q_i}$ corresponds to the relation $\widetilde{Q_i}(\mathrm{var}(\widetilde{Q_i}))$.

Table 1 OU-Schema and its instance

$\widetilde{Q_i}$	Variable $?x_1$	Variable $?x_2$	···
$[\widetilde{Q_i}]$	x_{1_1}	x_{2_1}	···
	x_{1_2}	x_{2_2}	···
	···	···	···

Given two query patterns $\widetilde{Q_i}$ and Q_i, as shown in Table 2, if there is a one to one map ν from var($\widetilde{Q_i}$) to var(Q_i), we can replace each variable $?\widetilde{x_i}$ in $\widetilde{Q_i}$ with $\nu(?\widetilde{x_i})$ in var(Q_i) and obtain $\nu(\widetilde{Q_i})$. If $\nu(\widetilde{Q_i}) = Q_i$, as shown in Table 2; they share the same instance by renaming attribute $?\widetilde{x_i}$ as $\nu(?\widetilde{x_i})$.

Table 2 Isomorphic query patterns and their instances

$\widetilde{Q_i}$	Variable $?\widetilde{x_1}$	Variable $?\widetilde{x_2}$	···
Q_i	Variable $?x_1$	Variable $?x_2$	···
$[Q_i]$	x_{1_1}	x_{2_1}	···
	x_{1_2}	x_{2_2}	···
	···	···	···

Definition 6 Isomorphic query patterns: two query patterns $\widetilde{Q_i}$ and Q_i are isomorphic, denoted as $\widetilde{Q_i} \equiv Q_i$ *iff* there is a one to one map ν from var($\widetilde{Q_i}$) to var(Q_i) so that $\nu(\widetilde{Q_i}) = Q_i$.

We also give the following definitions extended from the definition of isomorphic query patterns.

Definition 7 Given a set Γ of query patterns, a query pattern Q is isomorphically contained in Γ, denoted as $Q\widetilde{\in}\Gamma$, *iff* $\exists \widetilde{Q} \in \Gamma$, $Q \equiv \widetilde{Q}$.

Definition 8 A query pattern Q_i is semantically included by query pattern Q_j, denoted as $Q_i \subseteq Q_j$ *iff* there is a subset Q'_j of Q_j so that $Q'_j \equiv Q_i$.

Using the definition of isomorphic query patterns, given OU-Schema $\Gamma = \{\widetilde{Q_i}\}$, a query pattern Q can be divided into $(\bigcup_{i=1} Q_i) \cup (Q/\Gamma)$ where each $Q_i\widetilde{\in}\Gamma$ and $Q/\Gamma = Q - \bigcup_{i=1} Q_i$. We call Q/Γ the residue of Q with respect to Γ. Each $[Q_i]$ can be acquired from $[\widetilde{Q_i}]$ by renaming attribute $?\widetilde{x_i}$ in $[\widetilde{Q_i}]$ with $?x_i$ in $[Q_i]$ since Q_i and $\widetilde{Q_i}$ are isomorphic and $\nu(?x_i) = ?\widetilde{x_i}$. The residency $[Q/\Gamma]$ is returned by evaluating Q/Γ on a triple table using the RDF-3x query engine [4]. Finally $[Q]$ is obtained by natural joins of $[Q_i]$ and $[Q/\Gamma]$. That is $[Q] = [Q_1] \bowtie [Q_2] \cdots [Q_n] \bowtie [Q/\Gamma]$. The visual paradigm of our DC-Arc with OU-Schema is presented in Fig. 2. DC-Arc includes two components of one OU-Schema and one existing RDF query engine. The OU-Schema is built in front of the RDF query engine. A query pattern match consists of divide and conquer phases. In the divide phase, the query is guided to related tables in the OU-Schema. In the conquer phase, the instances of sub query patterns are joined together to return the answer to the query. Clearly, our DC-Arc needs to solve the following two key problems.

1. Efficiency problem: given OU-Schema at hand, how can a query be efficiently divided and therefore guided to related tables in Γ?
2. Effectiveness problem: how can an OU-Schema Γ be maintained so that for an incoming Q, the size $|Q/\Gamma|$ of Q/Γ is minimized while $\min\{|Q_i|\}$ is maximized?

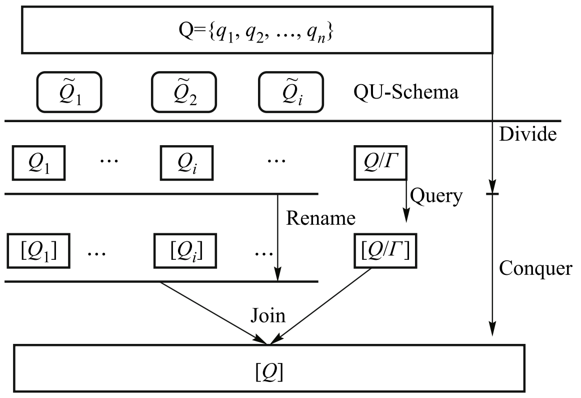


Fig. 2 OU-Schema and divide-conquer architecture

The goal of DC-Arc with OU-Schema is that instead of directly applying Q on G, we apply one small piece Q/Γ of Q on G while the rest are obtained from Γ without expensive long-chain join or union operations. Consequently, the performance and scalability of an RDF query is achieved.

5 Basic: isomorphism decision

In the DC-Arc, we need a method for dividing a query pattern into sub query patterns and relating them to the query patterns in a OU-Schema Γ. As the first step, it is necessary to decide whether two query patterns are isomorphic. Determining isomorphism plays two important roles in DC-Arc. Firstly, the isomorphic decision of two query patterns works to efficiently guide subquery patterns to related query patterns/tables in the OU-Schema. Secondly, it is necessary to employ isomorphic detection to extract query patterns and for an OU-Schema.

In practice, different users may issue query patterns with the same semantics while preferring different variable names. For example, one user issues the query $Q = \{(?b, \text{hasAuthor}, ?a), (?a, \text{wonPrize}, ?z)\}$, which queries the books written by an author who won a prize. Another user may write the same query in a different form $Q' = \{$(?book, hasAuthor, ?author), (?author, wonPrize, ?prize)$\}$. Formally, based on Definition 6, there are 3! one-one maps between var(Q) and var(Q'). An isomorphic decision between Q and Q' needs to check each and every map in the worst case. In this sense, it seems to be as difficult as the NP-hard graph isomorphism decision problem [27].

However, if we can sort triple patterns, for example in the following table, determining isomorphism can be reduced to checking between pairs $Q[i]$ and $Q'[i]$.

$Q[1]$	Q[2]
$(?b, \text{hasAuthor}, ?a)$	$(?a, \text{wonPrize}, ?z)$
$Q'[1]$	$Q'[2]$
(?book,hasAuthor, ?author)	(?author, wonPrize, ?prize)

A one-to-one map can be efficiently found between two serialized query patterns as shown in the example where $?b$ is mapped to ?book, $?a$ to ?author and $?z$ to ?prize. Since different variable names with the same semantics may be preferred by different users, it is not useful to define an order on V. However, a user is not free to choose non-variable components in U. An order can be defined on U, for example, in terms of lexical order as done in the above table. Our basic idea is to serialize query patterns by sorting them in terms of non-variable components in U. It is not difficult to obtain an isomorphic map between two serialized query patterns.

5.1 Order on triple patterns

Inspired by the above example, we define an order on set $U \cup V$.

Definition 9 Total order set $(\leqslant, U \cup V)$: 1) $\forall x, y \in U$, $x < y$ *iff* x precedes y in terms of lexical order; 2) $\forall x \in U, y \in V$, $x < y$; 3) where $?x$, $?y \in V$ are always in the same rank.

The total order on $U \cup V$ can be extended to triple pattern space P.

Definition 10 Total order set $(\leqslant, P)$: each triple pattern is ordered by (s, p, o) order where each component of a triple pattern is ordered by $\leqslant$ in Definition 9. Let $q_1 \sim q_2$ denote two triple patterns in the same rank.

For example, $(?b, \text{hasAuthor}, ?a) < (?a, \text{wonPrize}, ?z)$ because $?b$ and $?a$ are in the same rank, but hasAuthor < wonPrize in terms of lexical order.

However, it is not always possible to serialize triple patterns in a query pattern. To simplify our problem, we define a special case where triple patterns in a query can be strictly serialized.

Definition 11 Strict query pattern: a query pattern Q is strict if $\neg\exists q_1, q_2 \in Q, q_1 \sim q_2$.

As shown in Fig. 3, based on the order in Definition 10, there may be many triple patterns in the same rank. We use $Q[i]$ to denote the set of all triple patterns of Q in the ith rank. In the general case of Fig. 3(a), in order to make determine whether Q and Q' are isomorphic, we have to check all one-to-one maps between var($Q[i]$) and var($Q'[i]$). In this sense, the order we define is not helpful. It is desirable in the case of Fig. 3(b), where only one map exists between var($Q[i]$) and var($Q'[i]$). Fortunately, as shown in Fig. 3(c), we can divide a query pattern Q into sub-query patterns Q_i which are as desirable as those in Fig. 3(b). Of course, there is more than one choice to divide a query pattern as we will show in the later part of this section.

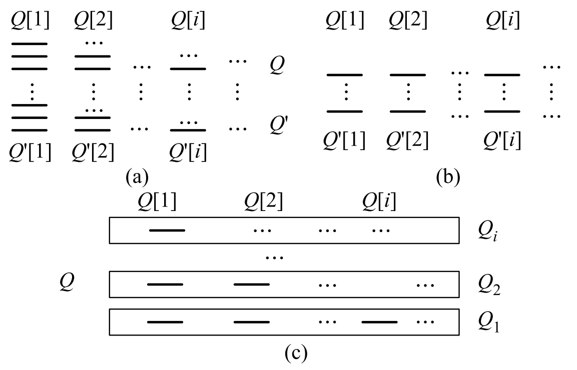


Fig. 3 Non-strict query patterns vs. strict query patterns. (a) Non-strict isomorphism decision; (b) strict isomorphism decision; (c) from non-strict to strict pattern

5.2 Determining isomorphism

In the special case of strict query patterns, two strict query patterns are sorted in terms of the order in Definition 10 and the pairs of variables in the same rank are compared. In order to speed up the test, we first normalize each strict query pattern as shown in Algorithm 1. After normalization, two isomorphic query patterns become the same. The details are shown in Algorithm 2. In the following table, we give two queries Q and Q', where $Q \equiv Q'$ since sNorm(Q) = sNorm(Q').

Lemma 1 If Q_1 and Q_2 are two isomorphic strict query patterns, $Q_1[i] \sim Q_2[i]$, where $Q[i]$ is the ith triple pattern in Q.

Theorem 1 Given two strict query patterns Q and Q', $Q \equiv Q'$ according to Algorithm 2 iff sNorm(Q) = sNorm(Q').

$Q[1]$	Q[2]
(?b, hasAuthor, ?a)	(?a, wonPrize, ?z)
sNorm⇂	sNorm⇂
(?1, hasAuthor, ?2)	(?2, wonPrize, ?3)
$Q'[1]$	$Q'[2]$
(?book, hasAuthor, ?author)	(?author, wonPrize, ?prize)
sNorm⇂	sNorm⇂
(?1, hasAuthor, ?2)	(?2, wonPrize, ?3)

Algorithm 1 sNorm(Q)
```
1: Sort $Q$;
2: if $Q$ is NOT a strict query pattern then
3:   return;
4: end if
5: for each $?x \in$ var($Q$) do
6:   if $?x$ is the ith variable appearing in the sorted $Q$ then
7:     replacing $?x$ with $?i$;
8:   end if
9: end for
10: Output sNorm($Q$);
```

Algorithm 2 sNorm(Q_1, Q_2)
```
1: Sort $Q_1$ and $Q_2$, respectively;
2: if a non-strict query pattern then
3:   return unknown;
4: end if
5: if sNorm($Q_1$) = sNorm($Q_2$) then
6:   return true;
7: else
8:   return false;
9: end if
```

In the special case of strict query patterns, the complexity of our decision algorithm is linear with respect to the length of the query pattern. In later sections, we will show it is powerful enough for OU-Schema guided evaluation and OU-Schema maintenance.

5.3 Processing non-strict query patterns

The main difficulty of non-strict query patterns is that we have no efficient methods for determining isomorphism. One solution is to divide a non-strict query pattern Q into the union of strict query patterns Q_i. That is, we transform $\bigcup Q[i]$ to $\bigcup Q_j$ as shown in Fig. 3(c), where set $Q[i]$ contains the triple patterns in the ith rank and Q_j is strict.

There is more than one choice to generate $\bigcup Q_j$ from $\bigcup Q[i]$. We provide an example shown in Fig. 4. Intuitively, query pattern Q_1 of {(?b, hasAuthor, ?a1), (?a1, wonPrize, ?z1), (?z1, inList, firtClass)} expresses that a book is written by an author who won firstClass prize. Hence, triples in Q_1 form a semantic cluster. So also does the query pattern Q_2 of {(?b, hasAuthor, ?$a2$), (?$a2$, wonPrize,?$z2$), (?$z2$, inList, secondClass)}. But how can a machine divide a general query pattern into the union of strict query patterns in a semantic way?

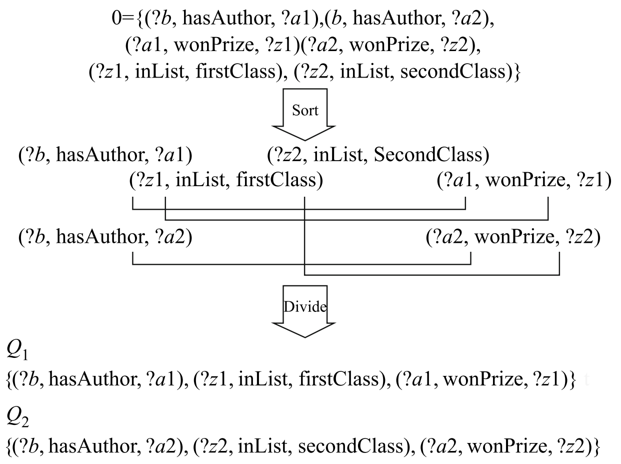

0={(?b, hasAuthor, ?a1),(b, hasAuthor, ?a2),
(?a1, wonPrize, ?z1)(?a2, wonPrize, ?z2),
(?z1, inList, firstClass), (?z2, inList, secondClass)}
Sort
(?b, hasAuthor, ?a1)
(?z2, inList, SecondClass)
(?z1, inList, firstClass)
(?a1, wonPrize, ?z1)
(?b, hasAuthor, ?a2)
(?a2, wonPrize, ?z2)
Divide
Q1
{(?b, hasAuthor, ?a1), (?z1, inList, firstClass), (?a1, wonPrize, ?z1)}
Q2
{(?b, hasAuthor, ?a2), (?z2, inList, secondClass), (?a2, wonPrize, ?z2)}

Fig. 4 From $\bigcup Q[i]$ to $\bigcup Q_j$

We use a query graph as the basic tool to generate strict query patterns. A query pattern Q is considered a query graph, where each triple pattern is a node and an edge exists between two triple patterns when they share the same variable. For example, an edge exists between (?b, hasAuthor, ?$a1$) and (?$a1$, wonPrize, ?$z1$) in Fig. 4 because they share the common variable ?$a1$. The goal is to divide the query graph into semantic clusters where each cluster is a strict query pattern. Triple patterns in a semantic cluster are highly coupled, say {(?b, hasAuthor, ?$a1$), (?$a1$, wonPrize, ?$z1$), (?$z1$, inList, firtClass)} in Fig. 4, where two semantic clusters are highly decoupled. We propose a greedy approach to dividing a query graph into strict sub-graphs with the aim of minimizing the maximum number of edges between two sub-graphs. Thus, it is assumed that triple patterns in the same semantic cluster are most coupled with the maximum number of edges while the number of edges between two semantic clusters is minimized.

The details of dividing non-strict query patterns are shown in Algorithm 3. Initially, one of the hottest nodes with the maximum degree in the query graph is selected into Q_i as the seed. Repeat choosing one of the nodes with maximum number of edges incident with triple in Q_i if Q_i is still strict after the selected node is inserted. Until no node can be inserted into Q_i, Q_i forms a semantic cluster. Then, on $Q - Q_i$, repeat the above precess to form new semantic clusters.

In Fig. 4, initially, both nodes of (?b, hasAuthor, ?$a1$) and (?b, hasAuthor, ?$a2$) have maximum degree. One of them can be selected as the seed to build a semantic cluster. If (?b, hasAuthor, ?$a1$) is selected into Q_1, (?b, hasAuthor, ?$a2$) is excluded from Q_1 since they are in the same rank $Q[1]$. In order to minimize the number of edges between Q_1 and $Q-Q_1$, a node with maximum edges connected to triples in Q_1 should be selected into Q_1 under the constraint that Q_1 is still strict. Therefore, (?$a1$, wonPrize, ?$z1$) is the only candidate to be added into Q_1 so that an edge between Q_1 and $Q - Q_1$ is eliminated. In the same way, (?$z1$, inList, firtClass)} is added and an edge is eliminated. Thereafter, no triple pattern can be added into Q_1 and Q_1 forms a semantic cluster where triple patterns in Q_1 are most coupled while Q_1 and $Q-Q_1$ are most decoupled. If $Q-Q_1$ is non-strict, the same procedure can be repeated to generate new strict query patterns.

Algorithm 3 strictDivide(Q)

```
1: while $Q$ is not strict do
2:   select one node with maximum degree into $Q_i$;
3:   repeat
4:     select one unmarked and unselected node $p$ with maximum edges incident with $Q_i$;
5:     if $Q_i \cup \{p\}$ is strict then
6:       insert $p$ into $Q_i$
7:     else
8:       mark $p$
9:     end if
10:  until no node can be inserted into $Q_i$
11:  report $Q_i$ as a semantic cluster;
12:  $Q = Q - Q_i$;
13:  unmark all nodes;
14: end while
```

Theorem 2 $Q = \bigcup Q_i$, where each strict query pattern Q_i is output by Algorithm 3 with Q as input.

Algorithm 3 generates the minimum number of strict query patterns while relations between strict query patterns are most decoupled in terms of min – max number of edges between strict query patterns.

Because there is more than one node with the maximum edges incident with the selected nodes, Algorithm 3 may generate different solutions from input Q. Therefore, it is possible that a common query pattern included in most queries can be divided and partially included in different strict query patterns. Therefore, the common query pattern is possibly excluded from an OU-Schema. In empirical experiments, we find this negative impact is not significant. However, in later sections, we will show the union of strict query patterns can significantly improve the performance of a query evaluation.

Algorithm 3 possibly leaves triple patterns in Q which do not share any common variables and which form a strict query pattern (Line 1). However, we observe that in our experiment, the chance of appearing in the abnormal strict query patterns is slim. Therefore, the chance of including the abnormal query patterns in Γ is even more unlikely.

In subsequent sections, each query pattern is assumed to be a strict query pattern unless stated otherwise.

6 Efficiency: OU-Schema guided evaluation

To divide a query pattern Q into $(\bigcup_{\widetilde{Q_i}\in\Gamma} Q_i) \cup (Q/\Gamma)$, we can directly call Algorithm 2 to decide whether $Q_i \widetilde{\in} \Gamma$, where $Q_i \subseteq Q$. However, that is expensive because we need to enumerate $Q_i \subseteq Q$ and check whether $\widetilde{Q_i} \in \Gamma$. Further, it involves repeatedly calling Algorithm 2 to decide whether $\widetilde{Q_i} \in \Gamma$ for each $\widetilde{Q_j}$ in Γ.

As the first choice, index techniques can be used to control the exponential number of function calls. Using the order on $\mathbb{P}$ in Definition 10, we can serialize triple patterns in a query pattern by sorting them. After serialization, a strict query pattern is viewed as an interval with two end points $\min Q$ and $\max Q$. Hence, the OU-Schema Γ consists of strict query patterns is considered a set of intervals. Given a query pattern Q, we need to search all intervals in Γ which correspond to the sub-intervals included by Q. However, it is not easy to design an efficient index structure to speed up such a search, which is demonstrated in [28,29] due to the nature of the partial order on Γ.

To surmount the problem of indexing a partial order set, we transform the problem of searching included intervals into the problem of computing the semantic intersections between a query pattern and a set of query patterns. First, we give the definition of a semantic intersection between two query patterns as follows.

Definition 12 Semantic intersection: given two query patterns Q and Q', $Q\overrightarrow{\cap}Q' = \{q \in Q | \exists q' \in Q', q \sim q'\}$.

After $Q\overrightarrow{\cap}\widetilde{Q_i}$ is obtained for each $\widetilde{Q_i} \in \Gamma$, the following lemma states that we can filter out unnecessary function calls to decide whether $\widetilde{Q_i} \subseteq Q$.

Lemma 2 Given two query patterns Q and Q', $Q\overrightarrow{\cap}Q' \equiv Q'$ iff $Q' \subseteq Q$.

In the following subsections, we first consider the problem of computing $Q\overrightarrow{\cap}\widetilde{Q_i}$ for each $\widetilde{Q_i} \in \Gamma$ in a batch with the help of index techniques. We then consider the problem of dividing Q with respect to Γ.

6.1 Batch computing semantic intersections

We design a B^+-tree like structure to speed up computing $Q\overrightarrow{\cap}\widetilde{Q_i}$ for each $\widetilde{Q_i}$ in Γ.

We consider Γ as $\bigcup_{\widetilde{Q_i}\in\Gamma} \widetilde{Q_i}$ and sort all contained triple patterns based on the order in Definition 10. In Fig. 5, each line segment represents a triple pattern and triple patterns in the same rank are placed in one column. We use the pair (q_i, ID_{q_i}) to denote triple patterns of the same rank, where ID_{q_i} contains all ids of query patterns which include a triple pattern in the same rank with q_i. Therefore, in Fig. 5, each column corresponds to a pair of (q_i, ID_{q_i}).

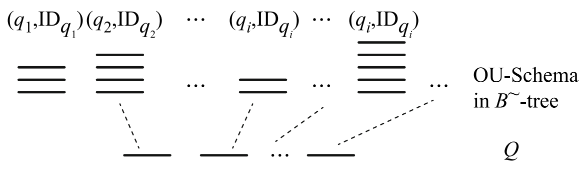

$(q_1,\mathrm{ID}_{q_1})(q_2,\mathrm{ID}_{q_2})$ ··· (q_i,ID_{q_i}) ··· (q_i,ID_{q_i})
OU-Schema in $B^{\sim}$-tree
Q

Fig. 5 OU-Schema storage and pattern division

In Fig. 5, given a query pattern Q, in the sorted list of $(q_1, \mathrm{ID}_{q_1})\ (q_2, \mathrm{ID}_{q_2}) \cdots (q_i, \mathrm{ID}_{q_i}) \cdots$ ordered by q_i, each triple pattern q_j in Q corresponds to (q_i, ID_{q_i}) if $q_j \sim q_i$. We handle triple patterns in Q one by one according to their order. For a triple pattern q_j in Q, we first locate its corresponding pair (q_i, ID_{q_i}). Then for each id in ID_{q_i}, if the set Q_{id} does not exist, we create it and insert q_j, otherwise we directly insert q_j. Thus, we have $Q_{\mathrm{id}} = Q\overrightarrow{\cap}\widetilde{Q}_{\mathrm{id}}$. All semantic intersections between Q and Γ are obtained in a batch.

Theorem 3 Given a query pattern Q and a set Γ of query patterns, in terms of the number of triple patterns in Q, there is a linear complexity algorithm which computes each $Q\overrightarrow{\cap}\widetilde{Q_i}$, where $\widetilde{Q_i} \in \Gamma$.

In order to maintain the sorted list of $(q_1, \mathrm{ID}_{q_1})\ (q_2, \mathrm{ID}_{q_2}) \cdots (q_i, \mathrm{ID}_{q_i})$ ordered by q_i shown in Fig. 5, the B^+-tree like structure $B^{\sim}$-tree is used so that it admits insertion and deletion. Another B^+-tree like structure $B^{\equiv}$-tree is used to store the OU-Schema Γ. Both trees are the augmented structure of B^+-tree.

The non-leaf node of $B^{\sim}$-tree is $\langle p_1, q_1, p_2, q_2, \ldots, p_i, q_i, \ldots\rangle$ following the same semantics of standard B^+-tree, where p_i is a pointer to nodes less than q_i. The leaf node is $\langle(q_1, \mathrm{ID}_{q_1}), (q_2, \mathrm{ID}_{q_2}), \ldots, (q_i, \mathrm{ID}_{q_i}), \ldots\rangle$, where each ID_{q_i} is the set of query pattern id*s*. Each query pattern with its ID in ID_{q_i} has a triple pattern in the same rank with q_i.

Similarly, all triple patterns are stored in the $B^{\equiv}$-tree which logically provides a sorted list of (ID, q) ordered by (ID, q) where ID is the identification of the query pattern containing q. In the $B^{\equiv}$-tree, the triple patterns with the same ID are clustered together in the sorted list.

In $B^{\sim}$-tree, each leaf node records the identifications of triple patterns in the same rank. $B^{\equiv}$-tree stores all query patterns. Therefore, $B^{\sim}$-tree is a compact structure of $B^{\equiv}$-tree by selecting a triple pattern as a candidate for those in the same rank.

Given a strict pattern Q and its identification ID, Algorithm 4 presents a sketch of their storage procedure.

Algorithm 4 sStore(Q, ID)

```
1: for $q_i \in Q$ do
2:   if $\exists(q, \mathrm{ID}_q)$, $q_i \sim q$ at a $B^{\sim}$-tree leaf then
3:     insert id into $\mathrm{ID}_q$ at $(q, \mathrm{ID}_q)$;
4:   else
5:     insert $(q_i, \{\mathrm{ID}\})$ into $B^{\sim}$-tree;
6:   end if
7:   insert $(\mathrm{ID}, q_i)$ into $B^{\equiv}$-tree;
8: end for
```

It is worth noting that the leaf nodes in $B^{\sim}$-tree contain field ID_{q_i} of set type which can occupy variable length of memory. In our implementation, a sorted list is used to store ID_{q_i} and a pointer to it is stored at the node. Therefore, each node can be held in a fixed length memory unit. Furthermore, the split/merge operations, as nodes are inserted/deleted, are the same as in standard B^{+}-tree.

6.2 OU-Schema guided evaluation of a query pattern

In this subsection, we present Algorithm 5 for dividing Q with respect to a given Γ. First, each Q_{id} is obtained with the help of index $B^{\sim}$-tree (Line 1). Given Q_{id} and $\widetilde{Q}_{\mathrm{id}}$ are strict query patterns, if $|Q_{\mathrm{id}}| = |\widetilde{Q}_{\mathrm{id}}|$ then it is possible that $Q_{\mathrm{id}} \equiv \widetilde{Q}_{\mathrm{id}}$ (Line 3). Algorithm 2 is called to decide whether $Q_{\mathrm{id}} \equiv \widetilde{Q}_{\mathrm{id}}$ (Lines 4–6). Finally, the algorithm outputs all marked Q_{id} as $Q_{\mathrm{id}}\widetilde{\in}\Gamma$. Therefore, $Q = (\bigcup_{Q_{\mathrm{id}}\widetilde{\in}\Gamma} Q_{\mathrm{id}}) \cup Q/\Gamma$.

Algorithm 5 isoDivide(Q, Γ)

```
1: Compute each $Q_{\mathrm{id}}$ using index $B^{\sim}$-tree;
2: for each $Q_{\mathrm{id}}$ do
3:   if $|Q_{\mathrm{id}}| = |\widetilde{Q}_{\mathrm{id}}|$ then
4:     if sIsom($Q_{\mathrm{id}}$, $\widetilde{Q}_{\mathrm{id}}$) then
5:       insert mark $Q_{\mathrm{id}}$; //as $Q_{\mathrm{id}}\widetilde{\in}\Gamma$;
6:     end if
7:   end if
8: end for
9: Output all marked $Q_{\mathrm{id}}$;
```

Theorem 4 Given a query pattern Q with respect to Γ, each query pattern $\widetilde{Q_i}\in\Gamma$ is output by Algorithm 5, where $Q_i \subseteq Q$.

The complexity of computing all $\widetilde{Q_{id}}\in\Gamma$ is linear, as is the complexity of determining whether two strict query patterns are isomorphic. The value of $\widetilde{Q_{id}}\in\Gamma$ is no more than the number of query patterns in Γ.

By applying Algorithm 5, Q is located and guided to its isomorphic query pattern $\widetilde{Q_i}$ in Γ. As shown in Table 1, $[\widetilde{Q_i}]$ is a flat table. By renaming attributes in $[\widetilde{Q_i}]$, $[Q_i]$ is obtained, which is also a flat table. It is tempting to perform merge-join operations in $[Q_1] \bowtie [Q_2] \bowtie \cdots \bowtie [Q_n] \bowtie [Q/\Gamma]$. Of course, we can apply a hash-join if merge-join is not available. The difficulty arises from the fact that we can not fix a sort of each $[Q_i]$ in advance so that each pair can be joined using a merge-join method. That is, we can not pre-sort $[\widetilde{Q_i}]$ in the order of unpredictable natural join attributes.

In our implementation, in addition to hash-joins of flat tables, we can also employ column-oriented storage [30] of each $[\widetilde{Q_i}]$ and therefore each column can be sorted. As a result, merge-join can be used to speed up $[Q_1] \bowtie [Q_2] \bowtie \cdots \bowtie [Q_n] \bowtie [Q/\Gamma]$.

6.3 Optimization

At the end of Algorithm 5, in a straightforward way, $[Q]$ can be obtained from $[Q_1] \bowtie [Q_2] \bowtie \cdots \bowtie [Q_n] \bowtie [Q/\Gamma]$. More precisely, Algorithm 5 only generates candidate views which can make contributions to a given query Q, but does not tell how they are incorporated into the query evaluation or evaluation optimization. From a plethora of issues in optimization under views [13], we consider the basic optimization problem of minimizing n, the number of candidate views. It is worth noting that the optimization we consider here does not necessarily lead to the optimum query evaluation shown in [13].

Let $\mathcal{S} = \{Q_i\}$, where Q_i is marked and output by Algorithm 5. The optimization problem we consider here can be reduced to the set cover problem [31]: $Q - Q/\Gamma$ is the input set, merge-join find the minimum number of sets in $\mathcal{S}$ such that their union includes $Q - Q/\Gamma$.

7 Effectiveness: OU-Schema

In this section, we define $\mathrm{cov}_{\Gamma}(Q) = (|Q| - |Q/\Gamma|)/|Q|$ to measure the effectiveness of Γ to Q under memory constraints.

In order to raise $\mathrm{cov}_{\Gamma}(Q)$ as much as possible under memory constraints, one solution is to select those long query patterns into Γ which are frequently included in users' queries. Following the definition of maximal frequent patterns in [32], we give the following definition.

Definition 13 Maximal frequent query pattern: Given threshold f (usually 0.001), a query pattern Q is called frequent if there are at least $n \times f$ query patterns which semantically include Q, where n is the number of issued query patterns so far. A frequent query pattern Q is called a maximal frequent query pattern if there is no frequent query pattern Q' which semantically includes Q.

At a glance, we can apply streaming algorithms [11,12] for discovering frequent patterns. An item corresponds to a triple pattern, and an itemset to a query pattern. Frequent itemsets mean frequent query patterns. However, neither false positive algorithm [11] or false negative algorithm [12] of frequent itemset algorithm can be directly applied. However, it is easy to decide whether one itemset contains another with respect to the semantics of set inclusion but it is not easy to decide whether $Q_i \subseteq Q_j$. Naively, it is necessary to enumerate subsets of Q_j and check whether they are isomorphic to Q_i, which is expensive.

We provide the sketch of our method in Algorithm 6, which consists of three parts: frequency count, pattern generation, and pattern deletion.

Algorithm 6 sFrequ(QueryStreams, Γ)

```
1: while a new $Q$ from streams do
2:     $Q = (\bigcup_{\mathrm{id}} Q_{\mathrm{id}}) \cup (Q/\Gamma)$ by calling Algorithm 5;
3:     for each $\widetilde{Q_{\mathrm{id}}} \in \Gamma$ do
4:         increase the count of id by 1;
5:     end for
6:     generate and insert new patterns;
7: if $\Gamma$ is too large then
8:         delete query patterns;
9:     end if
10: end while
```

• **Frequency count** First, the issued query pattern Q is divided into Q_Γ and Q/Γ by calling Algorithm 5, where $Q_\Gamma = \bigcup_{\widetilde{Q_i} \in \Gamma} Q_i$ (reference Section 6). Then, the count for $\widetilde{Q}_{\mathrm{id}}$ is increased by 1 where $Q_i \equiv \widetilde{Q}_{\mathrm{id}}$.

• **Pattern generation** It is difficult to process the uncovered part Q/Γ of query pattern Q. In frequent pattern discovery over data streams [11,12], all query patterns included in Q/Γ are enumerated and counted. In our case, it is not helpful to materialize short query patterns since they make fewer contributions to maximizing coverage under the memory constraints. To avoid enumerating short query patterns, we consider Q/Γ as a graph where each triple pattern in Q/Γ is a node and an edge exists between two nodes when they share the same variable. We employ two methods of generating query patterns: random generation and greedy generation.

In random generation, each triple pattern is randomly selected in either a uniform or skewed way. In the uniform way, each triple pattern in Q/Γ is selected with the same probability, whereas different triples with different probabilities in a skewed way. For example, nodes incident with more edges are selected with higher probability. Then, the selected triple patterns forms an atomic query pattern.

In greedy generation, The node incident with the maximum edges is selected and all nodes adjacent to the node are also selected. Then the selected nodes form the first atomic query pattern. Repeat the above selection on the unselected nodes to form new atomic query patterns.

Some of the atomic query patterns are combined in terms of set union to combined query patterns. Both atomic query patterns and all combined query patterns are inserted into Γ by calling Algorithm 4 with unique identifications. The count for a new inserted query pattern is initiated with $\epsilon \cdot n - 1$, where n refers to the nth query input into Γ as done in Lossy Count [11].

• **Pattern deletion** With the insertion of new query patterns, Γ becomes too large to be held in main memory. Based on Lossy Count in [11], each query pattern with count less than ϵn is deleted.

After deletion, query patterns in Γ can be still condensed. A query pattern Q_1 subsumes a query pattern Q_2 if $Q_2 \subseteq Q_1$. If $Q_2 \subseteq Q_1$, Q_2 is deleted from Γ. Therefore, Γ stores the maximum frequent query patterns at last. In our implementation, first, query patterns in Γ are sorted by the descending query pattern sizes. Second, from the head of the sorted list, each query pattern is chosen to try subsuming query patterns after it and the subsumed query patterns are deleted.

Following the similar proof in Lossy Count [11], we give the following theorem.

Theorem 5 Maximal frequent query patterns with frequency counts no less than $(f - \epsilon) \times n$ are kept in Γ in Algorithm 6, where $\epsilon < f/10$.

8 Empirical evaluation

8.1 Setting

RDF-3x [4], a RISC-style engine for RDF queries, is the baseline in our empirical evaluation. Because we get similar observations on various data sets used in [4], we only report our experiments on the data set Yago in this paper. The data set Yago contains 40 114 899 distinct triples and 33 951 636 distinct strings, consuming 3.1 GB as (factorized) triple dump. RDF-3x [4] needs 2.7 GB for all indexes and the string dictionary. Using Microsoft C++ 6.0, we implement our DC-Arc with OU-Schema. RDF-3x code [4] is open source. We run algorithms on DELL EMS01@2.66 GHz with 4 GB memory.

8.2 Query pattern generator

We use random walk to generate query patterns with parameters in Table 3. Two triples with common components are neighboring. The set $n(p)$ contains triples neighboring p. Given the set P of triples, $n(P) = \bigcup_{p \in P} n(p)$. The degree of a triple p is $|n(p)|$. The frequency of a URI is the number of triples in the data set which contain the URI.

Table 3 Parameters of queries

R/Z	S	L	V	D	F
Random/Zipf	Strict	Length	Variable	Degree	Frequency

A query pattern Q is generated in two phases. In the first phase, we randomly select one of the triples from data sets in Q as the seed. Repeat adding a triple from $n(Q)$ into Q in either a random way or a skewed way until the required properties of Q are met. In the second phase, we replace some components of triples with variables to obtain query patterns also in either a random way or a skewed way. `R` means the generator selects a triple from $n(Q)$ randomly, while `Z` in a skewed way. The selection is skewed either by the frequencies where triples containing URIs with high frequencies are selected with higher probability, or by the degrees where triples with higher degree are selected with higher probability. Therefore, `FZ/DZ` means the selection is skewed by frequency/degree. `RVn/FVn/DVn` means some components of the selected triples in the first phase are replaced with a variable from the n unique variables in a random or skewed way. In the skewed way, the chance of replacement is skewed by either URI frequency or triple degree.

For example, `R100S50L10DV5` means 100 query patterns per group are generated randomly. Among them, 50 query patterns are strict, the average length of the query patterns is 10 triple patterns, the average number of variables in a query pattern is 5. The probability of replacing components with variables is skewed by degree.

We observe that query patterns generated skewed by high degree can lead to a large number of intermediate results, whereas skewing by high frequency causes a large number of joins. Further, in the case of query patterns generated in a skewed way, OU-Schema is more effective at raising the coverage of a query pattern.

8.3 Testing

In Fig. 6, the two query patterns of type `Z100S100L5DV5` and `Z100S100L11DV5` are used as the base. Our DC-Arc is simply labelled DC. The coverage of Γ for a group of 100 query patterns is the average ratios of each $(|Q| - |Q/\Gamma|)$ to $|Q|$. The time dimension is the time cost for 100 queries. The label DC 30% stands for the performance of DC in the case of 30% coverage. In our experiment, we control the coverage by tuning the number of query patterns stored in Γ. Therefore, when the size of Γ is too large, we trigger deletion operations on Γ as shown in Section 7. It is worth noting that a long query pattern in Γ may cover the majority of a query.

In Fig. 6(a), we first study the time-efficiency of OU-Schema with respect to its coverage. The labels RDF-3x/5 tp and DC/5 tp stand for applying RDF-3x and DC on short patterns of type `Z100S100L5DV5`, respectively and the labels RDF-3x/11 tp and DC/11 for the long query patterns of type `Z100S100L11DV5`, respectively. Because we fix 100 query patterns as a group, the sizes of query responses remain the same as the coverage changes. In this way, we can observe how the coverage affects the query time without interference. The time gap between two methods becomes large as the coverage increases. For long query patterns, the gap is sharp because the flat tables of matches in Γ greatly reduce the number of joins in the case of processing long query patterns. Lines for two types of query patterns are sharply separated. Both methods spend much more time on long query patterns than short ones.

In Fig. 6(b), we choose the long query patterns of type `Z100S100L11DV5`. The coverage is fixed at 30%. The average number of matches for a group of 100 query patterns is listed along the x axis. In the 10 consecutive groups, DC achieves the sharp time advantage at the points where the number of matches is large. It looks surprising that DC achieves the greatest advantage at the Point 4 where the average 4×100 matches per 100 query patterns are returned. It is observed that at the Point 4, the intermediate results reach $3\,290 \times 100$ partial matches. In addition to the sizes of query answers, the query time heavily depends on the sizes of the intermediate results. DC achieves advantage in the case where the size of matches is small but the size of intermediate results is large.

In Fig. 6(c), each point on the x axis is the average length of 100 query patterns. It is observed that the time is affected by the average length of query patterns, where long patterns generally incur a high time cost. However, at Point 5, the time cost is less than at Point 3 because the matches at Point 5 are much fewer than at Point 3, this is compatible with the observations in Fig. 6(b). That is, the time is affected not only by the average length of query patterns but also the size of query pattern matches. At Point 9, DC achieves the greatest advantage mainly due to the large number of query pattern matches . At Point 11, only 290 matches are returned in our experiment and both methods use almost the same time.

In Fig. 6(d), based on `Z100S100L11DV5`, we slide the average number of unique variables occurring in query patterns from 3 to 9. The number of variables is increased by replacing some of the duplicated variables in triple patterns with different variables. The coverage of DC is set to 30%. The time cost increases as the number of unique variables increases in both methods. We also observe that the number of matches increases with the increasing number of unique variables, since more variables lead to fewer joins between triple patterns and therefore more matches. With more and more query pattern matches, the advantage of DC becomes even greater.

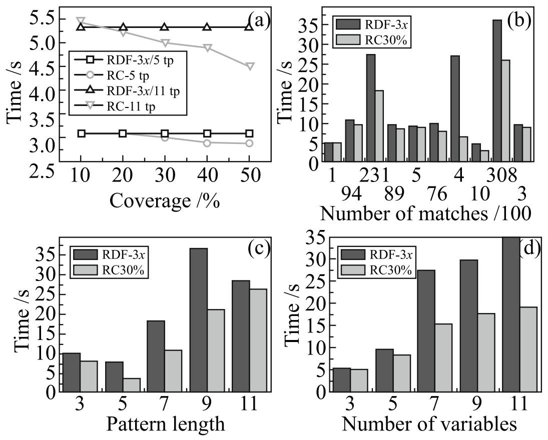


Fig. 6 Evaluation time per 100 queries. (a) Coverage; (b) matches; (c) pattern length; (d) variables

In Fig. 7, we study how the coverage changes in our greedy division method. Based on `R100S100L11DV5`, we generate query patterns of various types. In Fig. 7(a), the numbers of query patterns maintained in Γ are listed along the x axis. Our method for extracting frequent query patterns is effective in the case of skewed query patterns, which are labelled with the `Zipf Generation`. Especially after the point of 10 000 query patterns in Γ, the coverage is greatly improved for skewed query patterns. In Fig. 7(b), the percentages of non-strict query pattern of the query patterns are along x axis. It can be observed that the number of non-strict query patterns does not greatly affect the effectiveness of our method for processing non-strict query patterns. With more non-strict query patterns, some common query patterns can not be detected when our greedy generation method is used.

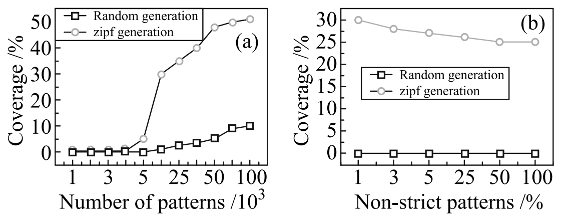


Fig. 7 Effectiveness. (a) Number vs. coverage; (b) Strictness vs. coverage

In our experiment, we find RDF-3x can lead to a machine crash when it evaluates some query patterns of more than nine 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. No machine crashes occur when DC is applied because Q/Γ, which is greatly shortened, is issued to RDF-3x.

The number of matches and the intermediate partial matches make a significant impact to query time. DC is more effective in the cases of long query patterns or large sizes of matches/partial matches. Our method of extracting frequent query patterns is memory-efficient and our greedy division for non-strict query patterns is effective.

9 Conclusion

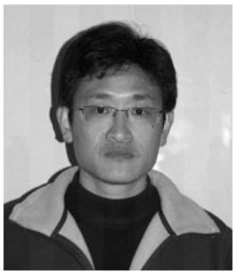
In this paper, we propose DC-Arc with OU-Schema where a query pattern is divided into two parts and conquered on flat tables in OU-Schema and vertical structures in backend storage. A lightweight approach is presented to guide a query to related flat tables and extract frequent common patterns semantically included in query streams. Based on the order we define on triple patterns, two B^+-tree like structures are designed to speed up query processing. Although our approach to serializing a query pattern possibly denies some chances of improving query performance, it is still efficient enough to circumvent the NP-hard nature of determining graph isomorpism and effective in our empirical evaluation. In future work, we will explore other methods for serializing query patterns so that query performance can be further improved. We will also study unified optimization techniques seamlessly integrated with those in backend query engines.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant Nos. 61003057 and 60973023). Besides, it was partially supported by the Natural Science Foundation of Jiangsu Province (BK2008354).

References

1. De Virgilio R, Giunchiglia F, Tanca L. Semantic Web Information Management. Heildelberg: Springer, 2010
2. Abadi D J, Marcus A, Madden S R, Hollenbach K. Sw-store: a vertically partitioned dbms for semantic web data management. The VLDB Journal, 2009, 18(2): 385–406
3. Arenas M, Gutierrez C, Pérez J. Foundations of RDF Databases. LNCS, 2009, 5689: 158–204
4. Neumann T, Weikum G. Rdf-3*x*: a risc-style engine for RDF. Proceedings of the VLDB Endowment, 2008, 1(1): 647–659
5. Weiss C, Karras P, Bernstein A. Hexastore: sextuple indexing for semantic web data management. Proceedings of the VLDB Endowment, 2008, 1(1): 1008–1019
6. Neumann T, Weikum G. Scalable join processing on very large RDF graphs. In: Proceedings of the SIGMOD '09. 2009, 627–640
7. Neumann T, Weikum G. The RDF-3*x* engine for scalable management of RDF data. The VLDB Journal, 2010, 19(1): 91–113
8. Chong E I, Das S, Eadon G, Srinivasan J. An efficient SQL-based RDF querying scheme. In: Proceedings of the VLDB '05. 2005, 1216–1227
9. Wilkinson K, Sayers C, Kuno H A, Reynolds D. Efficient RDF storage and retrieval in Jena2. In: Proceedings of the SWDB. 2003, 131–150
10. Chong Z, Qi G, Shu H, Bao J, Ni W, Zhou A. Open user schema guided evaluation of streaming RDF queries. In: Proceedings of the CIKM. 2010, 1281–1284
11. Manku G S, Motwani R. Approximate frequency counts over data streams. In: Proceedings of the VLDB'02. 2002, 346–357
12. Yu J X, Chong Z, Lu H, Zhou A. False positive or false negative: mining frequent itemsets from high speed transactional data streams. In: Proceedings of the VLDB'04. 2004, 204–215
13. Chaudhuri S, Krishnamurthy R, Potamianos S, Shim K. Optimizing queries with materialized views. In: Proceedings of the ICDE '95. 1995, 190–200
14. Fletcher G H, Beck P W. Scalable indexing of RDF graphs for efficient join processing. In: Proceedings of the CIKM'09. 2009, 1513–1516
15. Chen Y, Ou J, Jiang Y, Meng X. Hstar-a semantic repository for large scale owl documents. In: Proceedings of the ASWC. 2006, 415–428
16. Ma L, Wang C, Lu J, Cao F, Pan Y, Yu Y. Effective and efficient semantic web data management over DB2. In: Proceedings of the SIGMOD'08. 2008, 1183–1194
17. Matono A, Amagasa T, Yoshikawa M, Uemura S. A path-based relational RDF database. In: Proceedings of the ADC'05. 2005, 95–103
18. Battre D. Caching of intermediate results in DHT-based RDF stores. International Journal of Metadata, Semantics and Ontologies, 2008, 3(1): 84–93
19. Yan Y, Wang C, Zhou A, Qian W, Ma L, Pan Y. Efficient indices using graph partitioning in RDF triple stores. In: Proceedings of the ICDE'09. 2009, 1263–1266
20. Finkelstein S. Common expression analysis in database applications. In: Proceedings of the SIGMOD'82. 1982, 235–245
21. Papadomanolakis S, Ailamaki A. Autopart: automating schema design for large scientific databases using data partitioning. In: Proceedings of the SSDBM'04. 2004, 383
22. Chaudhuri S, Narasayya V. Self-tuning database systems: a decade of progress. In: Proceedings of the VLDB'07. 2007, 3–14
23. Godfrey P, Gryz J, Hoppe A, Ma W, Zuzarte C. Query rewrites with views for XML in DB2. In: Proceedings of the ICDE'09. 2009, 1339–1350
24. Chen D, Chan C Y. Viewjoin: efficient view-based evaluation of tree pattern queries. In: Proceedings of the ICDE. 2010, 816–827
25. Goh S T, Ooi B C, Tan K L. Demand-driven caching in multiuser environment. IEEE Transactions on Knowledge and Data Engineering, 2004, 16(1): 112–124
26. Sidirourgos L, Goncalves R, Kersten M, Nes N, Manegold S. Column-store support for RDF data management: not all swans are white. Proceedings VLDB Endowment, 2008, 1(2): 1553–1563
27. Wikipedia. Graph isomorphism problem, http://en.wikipedia.org/wiki/graph_isomorphism_problem
28. Hoel E G, Samet H. A qualitative comparison study of data structures for large line segment databases. In: Proceedings of the SIGMOD'92. 1992, 205–214
29. Hellerstein J M, Koutsoupias E, Papadimitriou C H. On the analysis of indexing schemes. In: Proceedings of the PODS'97. 1997, 249–256
30. Stonebraker M, Abadi D J, Batkin A, Chen X, Cherniack M, Ferreira M, Lau E, Lin A, Madden S, O'Neil E, O'Neil P, Rasin A, Tran N, Zdonik S. C-store: a column-oriented DBMS. In: Proceedings of the VLDB'05. 2005, 553–564
31. Kleinberg J. Algorithm Design. Person Education Inc., 2006
32. Yang G. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In: Proceedings of the KDD'04. 2004, 344–353

Weiwei Ni received his PhD in computer applications from Southeast University, Nanjing, China in 2005. He is now an associate professor of Southeast University and a member of the China Computer Federation. His research interests include data management and data mining.



Zhihong Chong received his PhD in Computer Software from Fudan University, Shanghai, China in 2006. He is now an associate professor of Southeast University. His research interests include data management and analysis.

Hu Shu is now a Master student of Southeast University. His research interests include graph data management and data mining.

Jiajia Bao is now a Master student of Southeast University. Her research interests include graph data management and data mining.

Aoying Zhou received his PhD from Fudan University, Shanghai, China in 1993. He is now a professor of East China Normal University. He is the winner of the National Science Fund for Distinguished Young Scholars supported by NSFC and the professorship appointment under Cheung Kong Scholars Program sponsored jointly by MoE and Li Ka Shing Foundation. His research interests include data management, data mining and data streams, and P2P computing.