# An Approach to RDF(S) Query, Manipulation and Inference on Databases

Jing Lu, Yong Yu, Kewei Tu, Chenxi Lin, and Lei Zhang

APEX Data and Knowledge Management Lab,
Department of Computer Science and Engineering,
Shanghai Jiao Tong University, Shanghai, 200240, China.
{robertlu, yyu, tkw, linchenxi, zhanglei}@apex.sjtu.edu.cn

**Abstract.** In order to lay a solid foundation for the emerging semantic web, effective and efficient management of large RDF(S) data is in high demand. In this paper we propose an approach to the storage, query, manipulation and inference of large RDF(S) data on top of relational databases. Specifically, RDF(S) inference is done on the database in advance instead of on the fly, so that the query efficiency is maximized. To reduce the cost of inference, two inference modes, the batch mode and the incremental mode, are provided for different scenarios. In both modes, optimized strategies are applied for efficiency purpose. In order to support efficient query and inference on the database, the storage schema is also specially designed. In addition, a powerful RDF(S) query and manipulation language RQML is provided for easy and uniform data access in a declarative way. Finally, we evaluate and report the performance on both query and inference of our approach. Experiments show that our approach achieves encouraging performance in million-scale real data.

## 1 Introduction

The semantic web is emerging as the next generation web, where web contents could be understood by machines. RDF(S) is a W3C standard for the formalization of information on web resources, laying one of the foundations of the semantic web. Thus effective and efficient management of RDF(S) data is a must, including storage, query, manipulation and inference.

Since scalability is of great concern in real-world applications, it is necessary to support the management of RDF(S) data in large volumes. Therefore, in this paper we propose an approach to the storage, query, manipulation and inference of RDF(S) on top of the relational database, which is the primary choice to manage large-volume data in practice.

A main feature of our approach is that, to maximize the query efficiency on inferred data, we store all the triples that are deduced from the RDF(S) closure computing, instead of calculating them on the fly. This is extremely imperative if a large data set is presented or the query is quite complex. One consequence of this decision is that we have to do inference on each manipulation, so as to maintain the consistence and completeness of the inferred data. Two modes

of inference, i.e. batch mode and incremental mode, are therefore developed to reduce the maintenance cost according to different manipulation scenarios. In addition, we introduce the Original Semantics Assumption and accordingly circumvent mutual dependency loops contained in the RDF(S) entailment rules while performing inference. The database schemas in our approach are elaborately designed so that not only query but also inference is well supported. We also introduce a language called RQML, which provides powerful and flexible query ability as well as manipulation ability. Finally, we report our performance evaluation, which shows that in a typical scenario the inference time complexity is around $O(n^2)$, and the query time complexity is around $O(n)$.

We have implemented our approach as the base layer of an integrated ontology engineering environment called ORIENT [1], which has been released at the IBM AlphaWorks website[1].

The rest of the paper is organized as follows. Section 2 presents the database schemas we adopted to store RDF(S) data. Section 3 discusses our inference algorithm on databases, including the batch and incremental mode. Section 4 introduces RQML, our RDF(S) query and manipulation language. The performance of our approach on query and inference is evaluated in Section 5. Finally we discuss the related work in Section 6 and conclude the paper in Section 7.

## 2 Database Schemas for Storage

In order to support efficient RDF(S) query and manipulation processing as well as RDF(S) inference (i.e. RDF(S) closure calculation), we carefully designed a relational database schema to store RDF(S) ontology. We followed the following principles in the design of the storage schema:

- For high-performance query, manipulation and inference, we trade storage space for speed.
- Although the storage schema should be able to handle arbitrary RDF(S) data, it must be optimized for normal data distribution and typical queries in ontology engineering scenarios.

Table.1 lists the database tables in the current storage schema design. The design of the tables are mainly influenced by and closely related to the RDF(S) closure computing algorithm, which is introduced in detail in the next section.

On the other hand, queries can also be efficiently carried out on these tables. Typical queries about class hierarchy, property hierarchy/domain/range and instance type can be answered quickly by directly querying the corresponding table. In addition to these tables for certain RDF schema triples, the `RDFLiteralInteger`, `RDFLiteralFloat`, `RDFLiteralBoolean` and `RDFLiteralString` tables are created to hold common data type literals and leverage native SQL data type comparison and calculation, which is required to support some kinds of queries. We also borrow Jena2's Property Tables design [2] in the `RDFUserProp` and `RDFProp*` tables to speed up the queries on certain user specified properties.

[1] http://www.alphaworks.ibm.com/tech/semanticstk

**Table 1.** Database tables

| Table Name | Table Description |
|---|---|
| RDFSubPropSubProp | This table holds all triples (?x, rdfs:subPropertyOf, rdfs:subPropertyOf). |
| RDFSubPropDomain | This table holds all triples (?x, rdfs:subPropertyOf, rdfs:domain). |
| RDFSubPropRange | This table holds all triples (?x, rdfs:subPropertyOf, rdfs:range). |
| RDFSubPropSubClass | This table holds all triples (?x, rdfs:subPropertyOf, rdfs:subClassOf). |
| RDFSubPropType | This table holds all triples (?x, rdfs:subPropertyOf, rdf:type). |
| RDFSubProp | This table holds all triples (?x, rdfs:subPropertyOf, ?y). |
| RDFDomain | This table holds all triples (?x, rdfs:domain, ?y). |
| RDFRange | This tables holds all triples (?x, rdfs:range, ?y). |
| RDFSubClass | This table holds all triples (?x, rdfs:subClassOf, ?y). |
| RDFType | This table holds all triples (?x, rdf:type, ?y). |
| RDFStatement | This table holds all triples (?x, ?y, ?z). |
| RDFResource | This table holds the URI or blank node ID for all RDF resources. |
| RDFLiteralInteger | This table holds all literals with type xsd:integer. |
| RDFLiteralFloat | This table holds all literals with type xsd:float. |
| RDFLiteralBoolean | This table holds all literals with type xsd:boolean. |
| RDFLiteralString | This table holds all literals with type xsd:string. |
| RDFTypedLiteral | This table holds all typed literals that are not contained in the above literal tables. |
| RDFPlainLiteral | This table holds all plain literals. |
| RDFUserProp | This table holds the names of user specified properties. |
| RDFProp* | These tables hold user specified property groups. |

The statement tables (i.e. the first eleven tables), which hold RDF(S) triples, contain a flag column which can take one of the following values: `EXPLICIT`, `DERIVED` and `SUSPENDED`. The value `EXPLICIT` and `DERIVED` indicate whether the statement is explicitly declared or is derived by inference. The value `SUSPENDED` is a temporary value which will be used in the process of inference.

Note that the RDF data are redundantly stored in the statement tables. For example, a statement like `(?x, rdf:type, ?y)` exists both in the `RDFType` table and `RDFStatement` table. In addition, as mentioned above, we store derived statements together with explicitly declared statements in the tables. This can also be seen as a kind of redundancy. However, this redundancy greatly facilitates and speeds up query processing.

- As derived statements are stored together with the explicitly declared statements, answering queries involving inference is almost as quick as answering queries without inference.
- Since the data contained in each table are complete by itself, the minimum number of tables are need to support a query. In this way, table join operations are greatly reduced. For example, the `RDFSubProp` table itself is able to support the property hierarchy queries.
- For most simple queries, only a single statement table is involved. As a result, a single SQL sentence can answer the query and hence we could simply utilize some SQL functions (e.g. ordering) provided by DBMS which are more efficient than those implemented by ourselves outside DBMS.

## 3 Inference on Databases

Reasoning on existing knowledge to discover implicit information is an important process on the Semantic Web. Common queries, such as "what are the (direct and indirect) sub-classes and instances of an exiting class" and "which instances have a certain relationship with a given instance", may all involve inference.

In real-world applications, most users wish to view a knowledge base at the semantic level, that is they want to query the derived data together with the explicitly declared ones. Currently in most RDF(S) management systems, the inference is not performed until it is imperative to answer a query. However, answering queries in this way might be time-consuming, especially when the data amount is large or the query is complex, which is the case in practice. Therefore, we choose to achieve better query performance at the cost of larger storage size. In other words, we choose to compute the complete RDF(S) closure and store all the derived RDF(S) statements in the database together with the explicit RDF(S) statements. When the knowledge base is modified, inference would be performed to maintain the consistency and completeness of the inferred data.

Our inference supports a core subset of the RDF(S) entailment rules including `rdf1, rdfs2` – `rdfs11, rdfs13` (as defined in the RDF Semantics document [3]). We call these twelve rules *the rule set* and the involved five RDF(S) properties (`rdfs:domain`, `rdfs:range`, `rdfs:type`, `rdfs:subPropertyOf` and `rdfs:subClassOf`) *the reserved property vocabularies*. This rule set is selected based on the entailment rules' importance and usage in common RDF ontology engineering scenarios.

In order to reduce the inference cost for different scenarios, two inference modes are provided, i.e. the batch mode and the incremental mode. In the batch mode, RDF(S) closure is not computed until the system is told to do so. This mode is suitable for batch update to the RDF data, since batch update may lead to a large amount of closure computation that would cost a long time. We design an optimized algorithm for this mode to compute the RDF(S) closure. In section 3.1, we will give the detailed description of this algorithm.

In the incremental mode, inference is performed whenever there are updates to the RDF data. The algorithm for the incremental mode is more like a forward-chaining closure computing method. However, since the forward-chaining closure computing method does not support retractions, we design an special algorithm for this purpose. We will give the detailed description in section 3.2.

### 3.1 Inference in the Batch Mode

A brute force method of calculating the closure is to repetitively grow the RDF knowledge base according to the rule set until a fix point is reached. This straightforward method, however, is very inefficient, especially when it is performed on a relational database. By analyzing the calculation process, we found that the inefficiency of the process mainly stems from the following problems:

**P1:** The two transitive predicates, `rdfs:subPropertyOf` and `rdfs:subClassOf`, give rise to repetitive calculation.

**P2:** The rules in the rule set, such as `rdfs7`, and the `rdfs:subPropertyOf` predicate create interdependent derivation relationships among the predicates. The iteration caused by such interdependency relationships is the main source of the inefficiency.

**P3:** On a relational database, the insertion of triples one by one is much less efficient than using batch SQL commands.

To better explicate first two problems, we present here a simplified graph of the predicates' derivation relationships caused by the rule set (Fig.1). Edge labels are names of the rules that cause the derivation relation. Two set of derivation relations are omitted from the graph:

**R1:** Rules `rdf1`, `rdfs2`, `rdfs3`, `rdfs4a` and `rdfs4b` may derive `rdf:type` from any predicate.

**R2:** Rule `rdfs7` may derive any predicate from any predicate due to `rdfs:subPropertyOf` relations.

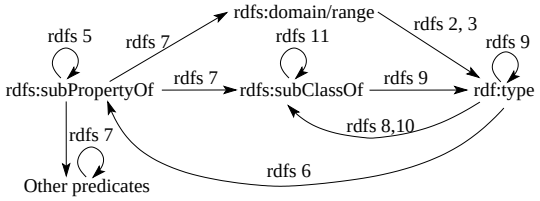


**Fig. 1.** Derivation Relationships among Predicates in The Rule Set

The self loops caused by `rdfs5`, `rdfs11`, `rdfs9` and `rdfs7` in the figure actually represent the problem P1. Without considering self loops, it is now clear from Fig.1 that `rdfs6, rdfs8, rdfs10` and R2 create big and complex loops in the derivation graph, which is just the cause of P2.

Since P2 is the main source of the inefficiency of the brute force method, now we first attack the problem P2. We find that, if some dependency relationships could be removed, then the graph could become a directed acyclic graph and the vertices (rules) could be topologically sorted, so the repetitive calculation in P2 can be reduced to just one pass. For this purpose, we now introduce the OSA (Original Semantics Assumption).

*Original Semantics Assumption* Let $\mathcal{A}$ be the set of RDF(S) axiomatic triples defined in the RDF Semantics document [3]. Let $\mathcal{T}$ be the closure of $\mathcal{A}$ under the rule set. Let $\Omega$ be the closure of the current RDF knowledge base under the rule set and $\mathcal{R} \subseteq \Omega$ be the set of all the triples whose subject is in the set {`rdfs:domain, rdfs:range, rdfs:type, rdfs:subPropertyOf, rdfs:subClassOf`}. The original semantics assumption supposes that $\mathcal{R} \subseteq \mathcal{T}$.

The assumption actually assumes that the RDF knowledge base does not strengthen the semantics of the five reserved RDF(S) property vocabularies.

They still keep their original meaning defined by the axiomatic facts. In most RDF(S) ontology engineering scenarios, this assumption is quite natural and can be satisfied because most applications does not require the change of the original RDF(S) semantics.

Under the OSA, the `rdfs6` and `rdfs10` rules

`rdfs6: (uuu rdf:type rdf:Property) → (uuu rdfs:subPropertyOf uuu)`
`rdfs10: (uuu rdf:type rdfs:Class) → (uuu rdfs:subClassOf uuu)`

become trivial. We can apply them only once after the `rdf:type` closure is obtained. We can be sure that no more triples can be obtained from the results of them. That is, they can not create loops in the derivation graph. Now let's look at the `rdfs8` rule:

`rdfs8: (uuu rdf:type rdfs:Class) → (uuu rdfs:subClassOf rdfs:Resource)`

Under the OSA, `rdfs:subClassOf` can not be other property's sub-property and `rdfs:Resource` can not be other class's sub-class. Therefore, this rule can be applied only once after the `rdf:type` closure is obtained. We can be sure that no more triples can be derived from the result of it. Hence this rule can not create loops in the derivation graph either. We can now safely remove the back lines caused by `rdfs6, rdfs8,` and `rdfs10` from Fig.1.

Finally, let's review R2. Because of the OSA, the five reserved vocabularies cannot be the sub property of other predicates. Hence the `rdfs7` rule can only create derivation relation from "other predicates" to the five reserved vocabularies in Fig.1. If R2 is now drawn on Fig.1, the only loop it can create is the one between the "other predicates" and the `rdfs:subPropertyOf`. However, this loop can be broken because the `rdfs:subPropertyOf` closure actually can be independently computed (step 2-4 of the algorithm described below).

Now, the only loops left in Fig.1 are the self loops. Except for these self loops, our algorithm can thus compute the entire closure in one pass.

In the following description, `?x, ?y, ?z` are used as variables to represent any resources or literals, with the restrictions that the predicates of a triple can only be URI references and the subjects of a triple cannot be literals.

1. Add all the RDF(S) axiomatic triples to the database.
2. Calculate the closure of all the triples with the form (`?x, rdfs:subPropertyOf, rdfs:subPropertyOf`)
   – This step can be done as follows. Let $\mathcal{P}$ be a set of resources that initially has `rdfs:subPropertyOf` as the only member. For any triples of the form (`?x,` $\mathcal{P}$, $\mathcal{P}$), add `?x` to $\mathcal{P}$. Repeat this rule until fix point. Then $\mathcal{P}$ should be the set of the possible sub properties of `rdfs:subPropertyOf`.
3. `(?x,` $\mathcal{P}$, `?y)` → `(?x, rdfs:subPropertyOf, ?y)`
4. Calculate the `rdfs:subPropertyOf` closure, that is, `(?x, rdfs:subPropertyOf, ?y)`, `(?y, rdfs:subPropertyOf, ?z)` → `(?x, rdfs:subPropertyOf, ?z)`
   – This step will be repeated until fix point.
   – After this step, we can obtain four sets $\mathcal{D}$, $\mathcal{R}$, $\mathcal{C}$ and $\mathcal{T}$, which represent the set of the sub properties of the reserved property vocabularies `rdfs:domain, rdfs:range, rdfs:subClassOf` and `rdf:type` respectively.

5. (?x, $\mathcal{D}$, ?y) → (?x, rdfs:domain, ?y)
6. (?x, $\mathcal{R}$, ?y) → (?x, rdfs:range, ?y)
7. (?x, $\mathcal{C}$, ?y) → (?x, rdfs:subClassOf, ?y)
8. Calculate the rdfs:subClassOf closure, that is, (?x, rdfs:subClassOf, ?y), (?y, rdfs:subClassOf, ?z) → (?x, rdfs:subClassOf, ?z)
   – This step will be repeated until fix point.
9. (?x, $\mathcal{T}$, ?y) → (?x, rdf:type, ?y)
10. (?x, ?y, ?z), (?y, rdfs:domain, ?a) → (?x, rdf:type, ?a)
11. (?x, ?y, ?z), (?y, rdfs:subPropertyOf, ?a), (?a, rdfs:domain, ?b) → (?x, rdf:type, ?b)
12. (?x, ?y, ?z), (?y, rdfs:range, ?a) → (?z, rdf:type, ?a)
13. (?x, ?y, ?z), (?y, rdfs:subPropertyOf, ?a), (?a, rdfs:range, ?b) → (?z, rdf:type, ?b)
14. (?x, ?y, ?z) → (?x, rdf:type, rdfs:Resource)
15. (?x, ?y, ?z) → (?z, rdf:type, rdfs:Resource)
16. (?x, ?y, ?z) → (?y, rdf:type, rdf:property)
17. (?x, rdf:type, ?y), (?y, rdfs:subClassOf, ?z) → (?x, rdf:type, ?z)
18. (?x, rdf:type, rdfs:Class) → (?x, rdfs:subClassOf, ?x)
19. (?x, rdf:type, rdfs:Class) → (?x, rdfs:subClassOf, rdfs:Resource)
20. (?x, rdf:type, rdfs:Datatype) → (?x, rdfs:subClassOf, rdfs:Literal)
21. (?x, rdf:type, rdf:property) → (?x, rdfs:subPropertyOf, ?x)
22. (?x, ?y, ?z), (?y, rdfs:subPropertyOf, ?a) → (?x, ?a, ?z)

After applying these steps , it can be proved that under the OSA, all triples which can be entailed by the rule set have been added to the database. We have also verified the correctness of our algorithm using W3C RDF test cases.

Now let's turn to the problem P1. In the two steps of computing the transitive closure on rdfs:subPropertyOf and rdfs:subClassOf, actually a Floyd [4] like algorithm is used for better efficiency. Take rdfs:subPropertyOf for example. For each property ?y, the rule "(?x, rdfs:subPropertyOf, ?y), (?y, rdfs:subPropertyOf, ?z) → (?x, rdfs:subPropertyOf, ?z)" is applied by table join to insert entailed triples into database in batches. This implementation has better performance than the two intuitive implementations as follows.

– One possible implementation is to apply the transitive rule directly on the database until fix point. On each iteration of applying the transitive rule, table join and batch insertion SQL command are used. The main drawback of such an implementation is that the less constrained self join of the RDFSubClass table will result in too many duplicate insertions, consuming too much time and memory when the data amount in the table is large.
– Another possible implementation is to read all the initial rdfs:subClassOf triples into the memory (if it is possible), perform the Floyd algorithm in the memory, and finally add the newly derived rdfs:subClassOf triples to the database. The main drawback of this implementation is that the one by one insertion of the resulting triples using SQL will consume much more time than using a similar batch insertion SQL command.

As to our Floyd like algorithm, the new triples are added to the database mainly in batches, while the self join of the RDFSubClass table is much more constrained, thus achieving better performance.

Finally, for the problem P3, on each rule application in our algorithm, we employ table join and batch insertion SQL command.

### 3.2 Inference in the Incremental Mode

In this section, we will describe the algorithm used in the incremental mode. In the incremental mode, every modification to the RDF(S) data will trigger computing the changes of the RDF(S) closure. The modification to the RDF(S) data can be regarded as appending some RDF triples and/or removing some RDF triples. We will discuss these two kinds of modification respectively.

When explicit triples are appended to the knowledge base, a simple forward-chaining closure computing is performed. The key point is that, according to the characteristic of the triple being appended, only a few rules in the rule set which may be relevant to the triple will be triggered. For example, if the predicate of the triple is not one of the RDF(S) reserved property vocabularies, only `rdf1, rdfs4a, rdfs4b` and `rdfs6` will be triggered.

When explicit triples are removed from the knowledge base, maintaining the consistency of the RDF(S) data is not as easy as when inserting triples. Since some triples derived from the removed triples may also be derived from some remaining triples, they could not be simply removed. However, examining one by one whether they can be derived from remaining triples is greatly time consuming. So we propose an algorithm which first removes all the suspect triples and then perform an incremental batch closure computing.

First, the triples to be removed are marked as `SUSPENDED` instead of being removed from the database immediately. Simultaneously, these triples are added to a queue for further processing.

Second, for each triple in the queue, the rules in the rule set that may be relevant to the triple are checked to see if there are `DERIVED` triples in the database that could be derived from the triple being processed.[2] If such triples exist, they are marked as `SUSPENDED` and are added to the queue. This process continues until the queue becomes empty. Actually, it is a Breadth-First Search.

Third, all the `SUSPENDED` triples are removed from the database. All the tables which contain `SUSPENDED` triples are marked as `DIRTY`.

Finally, an incremental batch closure computing, similar to the algorithm used in the batch mode, is performed. The difference is that, if a table is not marked as `DIRTY`, the steps that add triples to that table will be skipped.

## 4 Query and Manipulation through RQML

In order for both users and programs to query and manipulate RDF data in an uniform manner, here we define a declarative RDF Query and Manipulation Language (RQML). RQML is designed based on several previous RDF query languages such as RQL[5], RDQL[2] and SeRQL[6].

The queries in RQML are designed to be maximally syntax-compatible with RDQL, while at the same time borrow features like path expression from SeRQL.

[2] The triples that are marked as `SUSPENDED` are still regarded as valid triples in the database when the rules are being checked.

In addition, as literal comparison and calculation are frequently used in practice, RQML provides direct support of these features on several widely used XML literals. Such support is not available in most of the previous RDF query languages.

In addition to the above features, RQML queries support sorting on the result URI references or literals. Further, the implementation of RQML queries allows the query results to be read in a streaming fashion. Therefore with all these features, the use of RQML in practice can be quite scalable.

Being also an RDF manipulation language in addition to an RDF query language, RQML includes some necessary manipulation commands like INSERT, DELETE and UPDATE.

Here are some examples of RQML commands.[3]

- Example of literal processing: Find all the employees who is older than 35 and taller than 1.75 meter and whose stature is greater than 0.08 times his/her age.

```
SELECT ?p WHERE    (?p, <!http://employee/age>, ?a),
                   (?p, <!http://employee/height>, ?h)
           SUCHTHAT ?a > 35 AND ?h. > 1.75 AND ?h. > 0.08 * ?a
```

- Example of UPDATE command: Update the level of all the engineer employees with Linux certification to Senior Engineer.

```
UPDATE (?p, <!http://employee/level>, ‘‘Senior Engineer’’)
WHERE  (?p, <!http://employee/level>, ‘‘Engineer’’),
       (?p, <!http://employee/hasCert>, ‘‘Linux’’).
```

## 5 Query and Inference Performance

In this section we report the results of the experiments performed to empirically evaluate the performance of query and inference of our approach. The inference performance test is first presented followed by the query performance test.

Two data sets are used in the experiments. One is an artificial data set called “T57” that consists of only `rdfs:subClassOf` and `rdfs:subPropertyOf` relation triples that construct a class hierarchy tree and a property hierarchy tree. Both the two trees have a maximum height of 7 and a constant fan-out of 5. Another data set is “WN”, which is the RDF representation of WordNet[4]. All experiments are performed on a PC with one Pentium-4 2.4GHz CPU and 1GB memory running Windows XP Pro, using J2SDK 1.4.1 and Eclipse-SDK-2.1.1 connecting to a local machine DB2 UDB V8.1 Workgroup Server. The inference time is measured as the time cost to perform a full RDF(S) closure computing in the batch mode, starting from the database state of containing only the explicit triples. The query time is measured as the total time consumed from the sending of the query to finish fetching all the results from the database.

[3] Please refer to http://apex.sjtu.edu.cn/projects/orient/Documentation.htm#RQML for more examples.

[4] Available at http://www.semanticweb.org/library/

In our batch mode inference algorithm, the major component that determines the time complexity is the calculation of the transitive closure of sub properties and sub classes. The T57 data set is specifically designed to measure the empirical time complexity of it. By growing the two trees in T57 via adding one level of height each time, a series of growing data set for inference is got. The result is shown in Fig.2. The T57-1 line shows the relation between the inference time and the number of explicit triples. The T57-2 line shows the relation between the inference time and the number of triples after inference.

Different from T57, the WordNet data set has a very small RDF Schema with a very large set of instances and instance relations. The instance data in the four WordNet RDF files are sampled at the same speed. The number of sampled triples are multiplied by 5 each time and the triples from the four files are put together to get a series of growing data set for inference. The result is also shown in Fig.2. The WN-1 line shows the relation between the inference time and the number of explicit triples and the WN-2 line shows the relation between the inference time and the number of triples after inference.

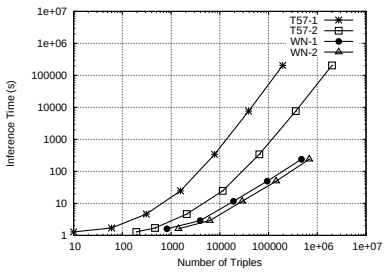


**Fig. 2.** RDF(S) Inference Performance

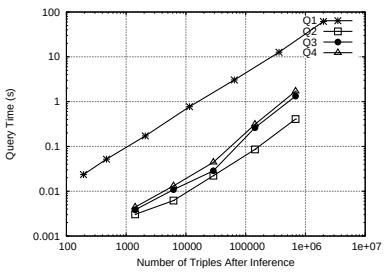


**Fig. 3.** RQML Query Performance

Both axes of Fig.2 are in log10 scale. The two T57 lines show a linear trend when the number of triples are large ($> 1000$). Linear regression analysis of the last four points at the end of each T57 line shows that the slopes are 1.87 and 1.75 for T57-1 and T57-2 respectively. This indicates approximately $O(n^{1.87})$ and $O(n^{1.75})$ time complexity. In theory, calculating transitive closure using Floyd algorithm has worst-case time complexity $O(n^3)$. When the algorithm is performed on a database, many factors of the RDBMS may further affect the performance. Combining the experiment result, we tend to empirically predicate that, when performed on a largely tree hierarchy ontology like T57 on a relational database, the time complexity of our batch inference algorithm is around $O(n^2)$.

Because the WordNet RDF data consists mostly of instance data, its number of inferred triples and inference time are much lower than T57 It, however, shows the same trend of linear relation in Fig.2. Linear regression analysis of the last four data points at the end of each WN line indicates approximately $O(n^{0.92})$ and $O(n^{0.93})$ time complexity. Similarly, we empirically expect that, when performed

on a largely instance data ontology like WordNet on a relational database, the time complexity of our batch inference algorithm is around $O(n)$.

We can see in Fig.2 that, in the experiments, the number of the triples after inference is in linear proportion to that of the explicit triples. If an RDF(S) ontology satisfies this property and the OSA, with characteristics between T57 and WN, we empirically estimate that its inference time complexity on a relational database is likely between $O(n)$ and $O(n^2)$. The $n$ here can represent either the number of the explicit triples or the number of the triples after inference.

The RQML query performance is tested on both T57 and WordNet data set with derived triples. We used the following four queries to test sub-class query, simple query, query with join and query involving literals:

```
Q1: SELECT ?X WHERE (?X, <rdfs:subClassOf>, [aClass])
Q2: SELECT ?X WHERE (?X, <wn:similarTo>, [randomAdjective])
Q3: SELECT ?Y WHERE ([randomNoun] <wn:hyponymOf>, ?X),
                    (?X, <wn:wordForm>, ?Y)
Q4: SELECT ?X WHERE (?X, <wn:wordForm>, ?Y) SUCHTHAT ?Y=[randomWordForm]
```

Q1 is performed on the T57 data set to obtain all subclasses of a given class. For each T57 data sample, and for each height of the tree hierarchy in that sample, Q1 is executed once using a class in that height. The query time of Q1 is then obtained as the average of the execution times. Q2, Q3 and Q4 are performed on the WordNet inferred data sets. The query time is averaged over 1000 query executions by randomly selecting a WordNet constant to replace the random constant in the above queries. The result is shown in Fig.3.

Both axes of Fig.3 is in log10 scale. Lines in Fig.3 are in linear trend, especially Q1. Linear regression analysis of the four lines shows approximately $O(n^{0.84})$, $O(n^{0.89})$, $O(n^{1.06})$, and $O(n^{1.05})$ time complexity for them. In theory, the worst-case time complexity of querying on one database table with indices is $O(n \log n)$. The actual query time also depends on the size of the result set. This test shows that the query time has a strong tendency of linear time $O(n)$ complexity and the query is executed quite speedy.

## 6 Related Work

There exist some other RDF(S) management systems such as Jena2 [2] and the Sesame system [6]. They also support storage and management of the RDF data on a relational database, but they only treat database as an alternative storage method. When answering queries, performing the inference on the fly is time consuming. In contrast, our approach performs efficient inference on database and stores all the derived triples to optimize the online query answering response.

The incremental inference algorithm used in our system is similar to the algorithm in [7]. The main difference is that, in [7], after marking some of the triples as `SUSPENDED`, these triples are examined whether they can be entailed from explicit triples. However, as this examining process requires accessing the database very frequently and fragmentally, such an approach is not as efficient as

it seems to be. On the other hand, although our algorithm may first remove some triples and then insert them back to the database, these removal and insertion operations are executed in a batch manner (that is, use less SQL commands to circumvent the problem P3 discussed in Sec.3). Thus, our algorithm could run faster than the one proposed by [7] in most cases.

Another inference algorithm that should be mentioned here is RETE [8]. As a generic rule-based forward-chaining algorithm, RETE is very efficient. However, as the RETE algorithm will consume lots of temporary memory in the inference procedure, it is not suitable for processing large-volume RDF data on databases.

## 7 Conclusion

In this paper, we present our approach on query, manipulation and inference of RDF data on relational databases. Different from most of the previous systems, the design of our approach aims at highly efficient query and inference with large-volume RDF data on relational databases. For query, we chose to store all the derived triples in database as well as the explicit ones. For inference, we carefully designed our algorithms for two inference modes, i.e. the batch mode and the incremental mode. We also defined a powerful RDF(S) query and manipulation language RQML, and presented the evaluation result of our approach.

## References


1. Zhang, L., Yu, Y., Lu, J., Lin, C., Tu, K., Guo, M., Zhang, Z., Xie, G., Su, Z., Pan, Y.: ORIENT: Integrate ontology engineering into industry tooling environment. In: Proceedings of the 3rd International Semantic Web Conference (ISWC2004). (2004)
2. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient RDF storage and retrieval in Jena2. In: Proceedings of the first International Workshop on Semantic Web and Databases (SWDB), Berlin,Germany (2003) 131–150
3. Hayes, P., McBride, B.: RDF Semantics. W3C Recommendation, W3C (2004) http://www.w3.org/TR/2004/REC-rdf-mt-20040210/.
4. Floyd, R.W.: Algorithm 97: Shortest path. Commun. ACM **5** (1962) 345
5. G.Karvounarakis, S.Alexaki, V.Christophides, D.Plexousakis, Scholl, M.: RQL: A declarative query language for RDF. In: Proceedings of the Eleventh International World Wide Web Conference (WWW02). (2002)
6. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying RDF and RDF Schema. In: Proceedings of the 1st International Semantic Web Conference (ISWC02). LNCS 2342 (2002) 54–68
7. Broekstra, J., Kampman, A.: Inferencing and truth maintenance in rdf schema. In: Proceedings of the First International Workshop on Practical and Scalable Semantic Systems (PSSS). (2003)
8. Forgy, C.: Rete: A fast algorithm for the many patterns/many objects match problem. Artificial Intelligence **19** (1982) 17–37