An Efficient SQL-based RDF Querying Scheme

Eugene Inseok Chong Souripriya Das George Eadon Jagannathan Srinivasan
Oracle
One Oracle Drive, Nashua, NH 03062, USA

Abstract

Devising a scheme for efficient and scalable querying of Resource Description Framework (RDF) data has been an active area of current research. However, most approaches define new languages for querying RDF data, which has the following shortcomings: 1) They are difficult to integrate with SQL queries used in database applications, and 2) They incur inefficiency as data has to be transformed from SQL to the corresponding language data format. This paper proposes a SQL based scheme that avoids these problems. Specifically, it introduces a SQL table function RDF_MATCH to query RDF data. The results of RDF_MATCH table function can be further processed by SQL's rich querying capabilities and seamlessly combined with queries on traditional relational data. Furthermore, the RDF_MATCH table function invocation is rewritten as a SQL query, thereby avoiding run-time table function procedural overheads. It also enables optimization of rewritten query in conjunction with the rest of the query. The resulting query is executed efficiently by making use of B-tree indexes as well as specialized subject-property materialized views. This paper describes the functionality of the RDF_MATCH table function for querying RDF data, which can optionally include user-defined rulebases, and discusses its implementation in Oracle RDBMS. It also presents an experimental study characterizing the overhead eliminated by avoiding procedural code at runtime, characterizing performance under various input conditions, and demonstrating scalability using 80 million RDF triples from UniProt protein and annotation data.

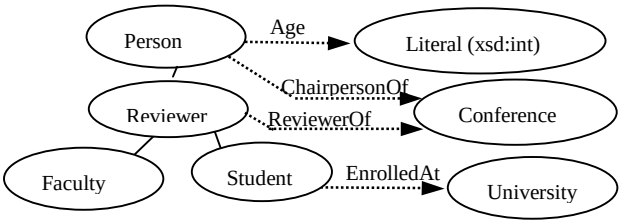
Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

1. Introduction

Resource Description Framework (RDF) [1] is a language for representing information (metadata) about resources in the World Wide Web. The resources are not limited to web pages but can also include things that can be identified on web. The specification of metadata in the generic RDF format makes it suitable for automatic consumption by a diverse set of applications.

The RDF data represented as a collection of <subject, property, object> triples, can easily be stored in a relational database. The paper addresses the issue of efficiently querying such RDF data. For querying RDF data, most approaches define yet another query language, which in turn issues SQL to process user requests. In contrast, this paper proposes a SQL-based scheme for querying RDF data. Specifically, it proposes an RDF_MATCH table function with the following functionality:

- The ability to search for an arbitrary pattern against the RDF data including inferencing based on RDFS [3] rules, and
- The ability to include a collection of user-defined rules as an optional data source.



Subject	Property	Object
IDBC2005	rdf:type	Conference
John	Age	24
John	rdf:type	Student
John	ReviewerOf	IDBC2005
Mary	rdf:type	Faculty
Mary	ChairpersonOf	IDBC2005

Figure 1: RDF data for reviewers model

To illustrate the basic functionality, consider RDF data about research paper reviewers. The RDF classes and the triple instances are shown in Figure 1. Assuming the RDF data is stored in the database as the model 'reviewers', user

can issue the following query to find reviewers who are students with age less than 25:

```
SELECT t.r reviewer
FROM TABLE(RDF_MATCH(
   '(?r  ReviewerOf  ?c)
    (?r  rdf:type    Student)
    (?r  Age         ?a)',
  RDFModels('reviewers'),
  NULL, NULL)) t
WHERE t.a < 25;
```

The various arguments to RDF_MATCH are as follows:

- The first argument captures the graph pattern to search for. It uses SPARQL-like syntax [13] and variables are prefixed with a '?' character.
- The second argument specifies the model(s) to be queried.
- The third argument specifies the rulebases (if any). Here the NULL argument indicates absence of rulebases.
- The fourth argument specifies user-defined namespace aliases (if any). Here the NULL argument indicates that no user-defined aliases are used, however default aliases such as rdf: are always available.

By processing RDF data using SQL the regular database tables can be queried in a single query along with RDF data. For example, a user can join results of RDF query with a traditional `employees` table say to find the `emailid` of the faculty reviewers:

```
SELECT t.r reviewer, e.emailid emailid
FROM TABLE(RDF_MATCH(
   '(?r  ReviewerOf  ?c)
    (?r  rdf:type    Faculty)',
  RDFModels('reviewers'),
  NULL, NULL)) t, employees e
WHERE t.r = e.name;
```

Providing RDF querying capability in SQL would enable applications to easily process domain-specific semantics stored as RDF data in a relational database. This becomes even more important especially in the context of semantic web applications, since RDF is an important building block of the semantic web [4]. Also, in future, if the vast amount of data stored in relational databases is made available as RDF triples [1], then the RDF_MATCH function can be used to query such data. Furthermore, applications that need to handle large volumes of metadata such as portals and e-marketplaces can also benefit from this functionality.

The proposed SQL-based RDF querying scheme involving the RDF_MATCH table function has been designed to meet most of the requirements identified in [5], including RDF Graph pattern matching, limiting resulting subgraphs, and returning results that may contain subgraphs from the graph entailed by input RDF graphs. It also handles RDFS and user-defined rules in a seamless manner along with the models.

With regards to implementation, the key aspects of our approach are as follows:

- The RDF_MATCH functionality is introduced as a SQL table function. Although this avoids kernel changes, the approach is not suitable for class of queries that return large result sets, where the overhead of returning result via the RDBMS table function infrastructure tends to dominate the query costs (see Section 4.1 for details). To circumvent this problem an extension to RDBMS table function infrastructure is implemented, which allows a rewrite of table function with a SQL query. With this extension, processing of RDF_MATCH table function query does not require any additional language run-time system other than the SQL engine.
- The RDF data triples are stored after normalization in two tables, namely IdTriples (triples in the identifier format) and UriMap (uri to identifier mapping). Other storage organizations are possible but are not considered in this paper. This would be addressed in future work. The storage representation supports handling literal of multiple datatypes as well as supports multiple representations of same literal value.
- The core implementation of RDF_MATCH query translates to a self-join query on IdTriples table. To efficiently execute this query, a set of B-tree indexes and materialized views are defined on the IdTriples and UriMap Tables.
- A class of materialized views called *subject-property matrix* materialized join views (SPMJVs) is introduced to avoid the inefficiency resulting from storing heterogenous data in the canonical triple format table. Also, the statistics collected on these SPMJVs serve as statistics for the corresponding portions of the triple table.
- Our approach relies on the RDBMS cost-based optimizer for optimizing the resulting query (that is, after rewrite of table function invocation). This approach has the advantage that RDBMS optimizer is leveraged. However, a shortcoming is that optimizer can generate sub-optimal plans. We plan to address this problem in future by enhancing the optimizer to better handle the class of self-join queries.
- Rulebases are in general handled by generating SQL queries, which may optionally involve table functions. Also we support the notion of indexing rulebases, which allows pre-computing and storing the data derived by applying rulebases to specified RDF models.

This scheme has been implemented in Oracle RDBMS using Oracle's table function infrastructure. In addition, the implementation uses Oracle's B-tree indexes, function-based indexes, and materialized views.

Performance experiments conducted using RDF data for WordNet, the lexical database for English language [8], and UniProt protein and annotation data [14] validate

the feasibility of this scheme and demonstrate that it scales well for large data sets.

The key contributions of the paper are:

- A SQL-based scheme for querying RDF data. No changes are made to SQL. Instead, RDF_MATCH table function is introduced. User can leverage all of the SQL capability to process the result of the RDF_MATCH table function.
- An efficient and scalable SQL based implementation of RDF_MATCH table function, including querying on data derived by applying rulebases..
- An extension to RDBMS table function infrastructure that eliminates bulk of the runtime overheads for class of table functions that can be expressed as a SQL query.
- A study characterizing RDF query performance as well as identifying overheads in various components.

1.1 Related Work

For querying RDF data, a number of query languages have been developed. This includes RDQL [9], RDFQL [10], RQL [11], SPARQL[13], SquishQL [1], and RSQL [15]. These are declarative query languages with quite a few similarities to SQL. However, the scheme proposed in this paper differs from all of the above in that it allows SQL itself to be used to query RDF data by introducing a table function. The main advantage of this SQL-based scheme is that it allows leveraging the rich functionality of SQL and efficiently combining graph queries with queries against traditional database tables.

With respect to handling rulebases, our scheme is quite similar to RDFQL where one or more data sources or rulebases can be specified.

With regards to implementation, query languages such as RQL and SquishQL try to push as much of the functionality as possible to underlying database by formulating SQL queries against tables storing RDF data. Our approach for implementing the RDF_MATCH table function is somewhat similar. However, it is tightly integrated with the SQL engine and with the table function SQL rewrite functionality further optimization is possible such as filter condition pushdown. Our approach uses materialized views to speed up queries on RDF data. This is in addition to the typical database indexes one can define on the tables storing RDF data.

1.2 Organization of the Paper

Section 2 describes key concepts of supporting RDF_MATCH based queries. Section 3 discusses the design and implementation of the RDF_MATCH function on top of Oracle RDBMS. Section 4 discusses an RDBMS table function infrastructure enhancement that can eliminate bulk of the mapping overhead for RDF_MATCH queries. Section 5 describes the performance experiments conducted using RDF data for WordNet and UniProt. Section 6 concludes with a summary and future work.

2. Key Concepts

This section gives the terminology used in the rest of the paper, outlines the requirements for querying RDF data, and describes how the SQL based RDF querying scheme meets these requirements.

2.1 Terminology

RDF can be used to capture domain semantics. The basic unit of information is a *fact* expressed as a <subject, property (predicate), object> *triple*. For example, the fact, 'John's age is 24', can be represented by <'John', 'age', '24'> triple. A collection of triples, typically pertaining to a domain or sub-domain, constitutes an *RDF model*.

Triples in a model can be classified as *schema* triples and *data* triples. Schema triples, specified using RDFS, describe the schema-related information (for example, <'age', 'rdfs:domain', 'Person'>), whereas data triples describe the instance data. Note that a triple's subject and property are always URIs while its object can be a URI or literal.

An RDF model is also referred to as *RDF graph*, where each triple forms a <property> edge that connects the <subject> node to the <object> node.

An RDF data set can optionally include one or more *rulebases*, each containing a collection of *rules*. A *rule* when applied to a model yields additional triples. An RDF model augmented with a rulebase is equivalent to the original set of triples plus the triples inferred by applying the rulebase to the model.

2.2 A SQL based RDF Querying Scheme

A table function is introduced to SQL as described below to satisfy most of the requirements described in [5] by a SQL-based RDF querying scheme.

RDF_MATCH Table Function: For data stored in a database, an RDF_MATCH table function is introduced with the ability to search for an arbitrary graph pattern against the RDF data, including inferencing based on RDFS and user-defined rules. The signature of the table function is as follows:

```
RDF_MATCH (
    Pattern       VARCHAR,
    Models        RDFModels,
    RuleBases     RDFRules,
    Aliases       RDFAliases,
  )
RETURNS AnyDataSet;
```

The first parameter captures the graph pattern to be matched. It is specified as a collection of one or more <Subject, Property, Object> triple patterns. Typically, each triple pattern contains some variables. Variables always start with a '?' character.

Among the remaining parameters, the first two specify a list of RDF models and (optional) rulebases, which

together constitute the RDF data to be queried. The last parameter specifies aliases for namespaces.

The result returned by RDF_MATCH is a table of rows. Each resulting row contains values (bindings) for the variables used in the graph pattern. Substituting the values in the graph pattern would identify the corresponding matching subgraph.

The exact definition of the result table, that is, the set of columns and their data types, varies depending upon the graph pattern used in an RDF_MATCH invocation. (Use of the `AnyDataSet` data type allows us to define RDF_MATCH with this flexibility.) Specifically, for each variable in a given graph pattern, the result table has a column with the same name as the variable (without the starting `'?'`). These columns are used for returning the lexical values for the corresponding variables. In addition, the result table has additional columns (of form `<variable>$type`) for returning data type information for each variable that may be bound to literals as well. Note that based upon current RDF restrictions (that subjects and predicates in triples must be URIs and not literals), only those variables that do not appear as subject and predicate components of triples in a graph pattern can be bound to literal values.

Example: Consider the following query (from Section 1) to find student reviewers who are less than 25 year old:

```
SELECT t.r reviewer, t.c conf, t.a age
FROM TABLE(RDF_MATCH(
  '(?r  rdf:type   Student)
   (?r  ReviewerOf ?c)
   (?r  Age        ?a)',
  RDFModels ('reviewers'),
  NULL, NULL)) t
WHERE t.a < 25;
```

The RDF_MATCH invocation returns the following table:

r	c	c$type	a	a$type
John	IDBC2005	URI	24	xsd:int

Note that, since variable `?r` appears as subject component, the value of column `r` is always a URI and hence there is no need for an additional column to return its data type.

SQL constructs may be used to extend the above query, to do aggregation, grouping, and ordering, for example:

```
SELECT t.c conf,
       COUNT(*) row_count, AVG(t.a) avg_age
FROM TABLE(RDF_MATCH(……)) t
GROUP BY t.c
ORDER BY avg_age;
```

Though the above examples show querying RDF data only, users can also query the associated RDF schema, for example, to obtain domains and ranges for a property. Thus, the key benefits of using a table function for querying RDF data is that the standard SQL constructs can be used for further processing of the results. This includes iterating over the results, constraining the results using WHERE clause predicates, grouping the results using GROUP BY clause, sorting the results using ORDER BY clause, and limiting the results by using the ROWNUM clause. Also, the SQL set operations can be used to combine result sets of two or more invocations of RDF_MATCH. With the table function SQL rewrite functionality discussed in Section 4.2 the optimizer will be able to optimize the whole SQL query including filter condition pushdown.

Rule and Rulebases: A rule is identified by a name and the rulebase to which it belongs. A rule consists of a left hand side (LHS) pattern for the antecedents, an optional filter condition that further restricts the subgraphs matched by the LHS, an optional list of namespace aliases, and a right hand side (RHS) pattern for the consequents. For example, the rule that "chair person of a conference is also a reviewer of the conference" is represented as follows:

```
('rb',                      -- rulebase name
 'ChairpersonRule',         -- rule name
 '(?r  ChairPersonOf  ?c)',-- LHS pattern
 NULL,                      -- filter condition
 NULL,                      -- aliases
 '(?r  ReviewerOf  ?c)')    -- RHS pattern
```

The following query will return both `John` and `Mary` as reviewers. The latter is implicitly inferred by applying rulebase `rb` to the `reviewers` model.

```
SELECT t.r reviewer
FROM TABLE(RDF_MATCH(
  '(?r  ReviewerOf ?c)',
  RDFModels ('reviewers'),
  RDFRules('rb'), NULL)) t;
```

A user can create rulebases and add rules by using APIs. Once the rulebases are created and populated, they can be specified in a RDF_MATCH query. Note that the RDFS rulebase (named `rdfs`) is created by the system and is implicitly available for users.

3. Design and Implementation

This section describes the design and implementation of the SQL-based RDF Querying Scheme. This scheme is implemented on top of Oracle RDBMS. Although the description here assumes Oracle RDBMS, the scheme can be supported in any RDBMS that supports table functions, materialized join views, and B-tree indexes.

3.1 RDF Data Storage and Multiple Data Type Hanlding

The RDF data must be stored compactly and the storage format should be suitable for efficient query processing. In our scheme, RDF data is stored (after normalization) in two tables: `IdTriples (ModelID, SubjectID, PropertyID, ObjectID, …)` and `UriMap (UriID, UriValue, …)`. This normalization is critical because URIs (or literals) are typically repeated. Also, it enables efficient query processing due to the compact size. Given an RDF triple, its three URIs (or literals) are first mapped to corresponding identifiers using table UriMap. If no mapping is found for a URI (or literal), a new unique UriID is generated and the new mapping is

inserted into the UriMap table. A tuple comprising the ModelID (for the RDF model) and the three UriIDs is then stored into the IdTriples table.

A user view is created on the underlying tables holding RDF data, which presents only selective portions (at model granularity) of the RDF data to the users based on their privileges. Also, the RDF_MATCH function is executed with invoker's privileges. Thus, this scheme limits each invoker's access via RDF_MATCH query to only the appropriate portion of the RDF data.

Typed literals are stored in the UriMap table with their type. To support matching between multiple representations for the same value, such as the integer 123 and the float 12.3E+1, each literal is mapped to a canonical literal. Literals that represent the same value will map to the same canonical literal, and a literal may be its own canonical literal. The canonical literal ID (which is used when joining on the object column) and the exact literal ID (which is used when returning the object to the user) are both stored in IdTriples. For simplicity, queries in this paper are written as if there was a single objectID column in IdTriples.

The first literal entered for a value becomes the canonical literal. To support mapping other equi-valued literals to this canonical literal, there is a flag in UriMap to indicate that the literal is a canonical literal. Further, the pre-defined datatypes are partitioned into families, where all types in a family are associated with a single value space. For example, float and integer types both belong to the numeric family. For each type family, there is a function to convert the UriMap lexical representations into a canonical form, such as a native database type. A function-based index for this purpose is defined on UriMap, so a canonical form can efficiently be mapped to the corresponding canonical literal during querying.

3.2 RDF_MATCH Table Function

The RDF_MATCH functionality is implemented as a SQL table function using Oracle's table function interfaces [6].

RDF Query Processing

- Compile Time Processing: At compile time, the form of the table result, namely the set of columns is determined. The kernel passes information regarding the columns referenced in the outer SQL query to the table function. This allows for optimization of table function queries based on columns referenced in the SQL query containing table function invocation.

- Execution Time Processing: Based on the input arguments, namely, pattern, models, rules, and aliases, a SQL query is generated against the IdTriples and UriMap tables. Figure 2 shows the various layers of implementation. There are two types of implementation: conventional procedural processing (discussed in Section 4.1) and a new declarative rewrite-based processing (discussed in Section 4.2). Due to overheads with the procedural implementation, RDF_MATCH function uses the rewrite-based implementation.

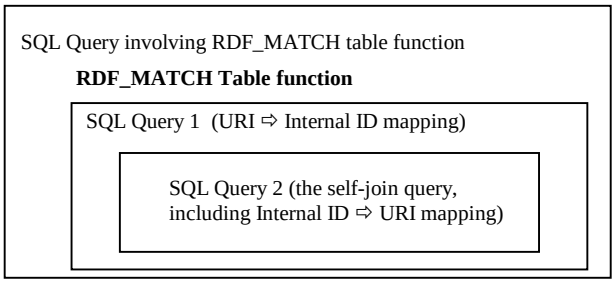


Figure 2: RDF_MATCH Implementation Overview

Consider as an example use of the RDF_MATCH table function in the following SQL query:

```
SELECT t.r reviewer, t.c conf, t.a age
FROM TABLE(RDF_MATCH(
     '(?r  ReviewerOf  ?c)
      (?r  rdf:type     Student)
      (?r  Age          ?a)',
……
```

First, aliases specified (if any) are substituted with the namespaces to expand all alias-encoded URIs. Next, the URIs and literals, such as 'ReviewerOf', 'rdf:type' are converted into UriIDs using lookups on the UriMap table:

```
FROM IdTriples t1, IdTriples t2. IdTriples t3
WHERE t1.PropertyID = 14 AND t2.PropertyID = 11
 AND  t2.ObjectID = 4    AND t3.PropertyID = 29
```

Then a self-join query is generated based on matching variables across triples (e.g. '?r') in the pattern:

```
WHERE … t1.SubjectID = t2.SubjectID AND
        t2.SubjectID = t3.SubjectID
```

Next, the internal IDs are joined with the UriMap table to generate the join result in the URI (and literal) format:

```
SELECT u1.UriValue r, u2.UriValue c,
       u2.Type c$type, u3.UriValue a,
       u3.Type a$type
FROM …… UriMap u1, UriMap u2, UriMap u3
WHERE …… t1.SubjectID = u1.UriID AND
         t1.ObjectID =  u2.UriID AND
         t3.ObjectID =  u3.UriID
```

Note that 'r' is a URI, so there is no type associated, whereas 'c' and 'a' have a date type (`c$type, a$type`) associated.

The models argument is used to restrict the IdTriples table based on the corresponding model identifiers in the above self-join SQL query. After the transformation phase, the generated single SQL query is optimized and executed to obtain results.

In addition to the table function arguments, kernel implicitly provides information regarding the columns (which correspond to the variables in the graph pattern) referenced elsewhere in the original SQL query. The RDF_MATCH implementation has been optimized to compute values only for these columns. This avoids additional joins with UriMap table to get the corresponding UriValue. In general, a query with n triple

pattern and m variables will result in a query with (n+m-1) joins, assuming m variables are projected (hence m joins with UriMap table). Experiment IV (Section 5.6) demonstrates the performance benefits of this optimization.

The rules argument contains a list of rulebases to be applied to infer new triples. This is discussed below.

Rule Processing: To handle rules, the RDF_MATCH function replaces references to the IdTriples table in the generated SQL with subqueries or table functions that yield the relevant explicit and inferred triples. Subqueries are used whenever the required inferencing can be done conveniently within a SQL query (i.e., without explicitly materializing intermediate results). These subqueries generally take the form of a SQL UNION with one UNION component for each rule that yields a relevant triple, plus one component to select the explicit triples. Table functions will be used when the subquery approach is not feasible.

To support the RDFS inference rules, we must compute a transitive closure for the two transitive RDFS properties: rdfs:subClassOf (rule rdfs11) and rdfs:subPropertyOf (rule rdfs5). In Oracle RDBMS, these transitive closures can be computed using hierarchical queries with the START WITH and CONNECT BY NOCYCLE clauses. Note that CONNECT BY NOCYCLE queries can handle graphs that contain cycles by generating the row in spite of the loop in user data. The remaining RDFS rules can be implemented with simple SQL queries.

To ensure that RDFS inferencing can be done within a single SQL query, the user is prohibited from extending the built-in RDFS vocabulary. This means, for example, that there cannot be a property that is a sub-property of the rdfs:subPropertyOf property, nor can there be a user-defined rule that yields rdfs:domain triples.

User-defined rules can be classified as follows based upon the extent of recursion, if any, in the rule:

- Non-recursive rules: The antecedents cannot be inferred by the given rule, or any rule that depends on the given rule's consequents.
- Simple recursive rules: These rules are used to associate transitivity and symmetry characteristics with user-defined properties.
- Rules that use arbitrary recursion unlike the other two categories.

Non-recursive user-defined rules can be evaluated using SQL (join) queries by formulating the FROM and WHERE clauses based upon the antecedents and the SELECT clause based on the consequents of the rule so as to return the inferred triples. Note that the triples that match the antecedents of a user-defined rule could themselves be inferred, so the FROM clause may reference subqueries to find inferred triples. The ChairpersonRule given in Section 2.2 would translate into SQL as follows:

```
SELECT ...
FROM (
-- (?x ChairpersonOf ?c) => (?x ReviewerOf ?c)
 SELECT t1.SubjectID, 14 PropertyID, t1.ObjectID
 FROM IdTriples t1
 WHERE t1.PropertyID = 56
 UNION
-- explicit ReviewerOf triples
 SELECT t1.SubjectID, t1.PropertyID, t1.ObjectID
 FROM IdTriples t1
 WHERE t1.PropertyID = 14
) t1;
```

Simple recursive rules involving transitivity and symmetry can be evaluated as follows. Symmetry can be easily handled with a simple SQL query. However, handling transitivity with a single SQL query requires some type of hierarchical query (e.g., using the START WITH and CONNECT BY NOCYCLE clauses in Oracle RDBMS), as in the case of transitive RDFS rules.

Suppose the user's query is:

```
...
RDF_MATCH(
      '(?a  rdf:type    Male)
       (?a  AncestorOf  ?b)',
...
```

There is a user-defined rule to make AncestorOf transitive, and for simplicity we assume that the RDFS rulebase is not used. So after translation we have a join between IdTriples (for the rdf:type triple) and a subquery which computes the transitive closure using CONNECT BY (for AncestorOf):

```
SELECT ...
FROM IdTriples t1, (
  SELECT DISTINCT
     CONNECT_BY_ROOT(t1.SubjectID)  SubjectID,
     t1.PropertyID, t1.ObjectID
  FROM IdTriples t1
  START WITH t1.PropertyID = 43
  CONNECT BY NOCYCLE t1.PropertyID = 43 AND
             PRIOR ObjectID = SubjectID
  ) t2
WHERE t1.PropertyID = 11 AND t1.ObjectID = 17
      AND t1.SubjectID = t2.SubjectID;
```

The third class of rules involving arbitrary recursion is the most complicated, and it has not been addressed in the current implementation. These rules will be evaluated using table functions, because an unknown number of passes over the intermediate results are required to find all inferred triples.

3.3 Speeding up RDF_MATCH Queries

The speed up is achieved by creating materialized join views (MJVs) and creating appropriate B^+-tree indexes on them, and indexing RDF data and rulebases. Each of these is described in detail below.

Generic Materialized Join Views: The query generated by RDF_MATCH table function involves a self-join of IdTriples table if the same variable is used in more than one triple of the search pattern. Depending on how many triples are specified, a multi-way join needs to be executed. Since the join cost is a major portion of the total

processing time, materialized join views can be defined to speed up RDF_MATCH processing. The row size of IdTriples table is small and hence the materialized join view can be a good candidate for reducing the join cost. In general, six materialized two-way join views, namely joins between `SubjectID-SubjectID`, `SubjectID-PropertyID`, `SubjectID-ObjectID`, `PropertyID-PropertyID`, `PropertyID-ObjectID`, and `ObjectID-ObjectID` can be defined as long as the storage requirement is met. Most useful materialized join views for typical queries, however, are joins between `SubjectID-SubjectID`, `SubjectID-ObjectID`, and `ObjectID-ObjectID`. Note that the individual materialized join views could be created for a subset of data based on the workload characteristics.

The materialized join views are incrementally maintained on demand by the user using the `DBMS_MVIEW.REFRESH API`. A procedural API is provided to analyze IdTriples table to estimate the size of various materialized views, based on which a user can define a subset of materialized views.

Subject-Property Matrix Materialized Join Views: To minimize the query processing overheads that are inherent in the canonical triples-based representation of RDF, subject-property matrix based materialized join views can be used. These materialized views can be designed using the following basic ideas:

- For a group of subjects, choose a set of single-valued properties that occur together. These can be *direct* properties of these subjects or *nested* properties. A property p_1 is a direct property of subject x_1 if there is a triple (x_1, p_1, x_2). A property p_m is a nested property of subject x_1 if there is a set of triples such as, (x_1, p_1, x_2), …, (x_m, p_m, x_{m+1}), where m >1. For example, if we have a set of triples, (John, address, addr1), (addr1, zip, 03062), then zip is a nested property of John.
- Create a (subject-property matrix) materialized join view each of whose rows contains values of these properties for a subject in the group.

Query performance can be improved significantly through the use of such materialized join views because a number of joins can be eliminated. For example, Table 1 shows a sample RDF data and Table 2 shows a matrix materialized join view created for subjects who are `Students` with their direct property `age` and nested property `city` (named in the view as `studiesAt` to denote the city where his/her university is located).

This subject-property matrix can be exploited by an RDBMS optimizer to process an RDF query using the following query pattern to retrieve the `age` and `studiesAt` info for each `student`:

```
'(?r  rdf:type    Student)
 (?r  enrolledAt  ?u)
 (?r  age         ?a)
 (?u  city        ?city)'
```

and retrieving values of variables `?r`, `?city`, and `?a`.

Table 1: Student Info RDF Data

Subject	Property	Object
John	rdf:type	Student
John	EnrolledAt	Univ1
John	Age	24
Pam	rdf:type	Student
Pam	EnrolledAt	Univ2
Pam	Age	22
Univ1	UnivName	NYU
Univ1	City	New York
Univ2	City	Los Angeles

Table 2: Student Matrix

Subject	StudiesAt	Age
John	New York	24
Pam	Los Angeles	22

This query will normally require a 4-way self-join on the IdTriples table (leaving out the conversion between IDs and URIs, for simplicity). However, by using the matrix in Table 2, the query can be processed by simply selecting all the rows from the materialized join view. Thus, self-joins can be completely eliminated in this case. This can lead to significant speed-up in query processing.

In general, for the type of queries shown above a query requiring an `n`-way join could potentially be processed using a matrix with `m`-properties with `(n – m)` joins.

In typical usage of such matrices, each subject in the group will have one value for each of the chosen properties. Usage may involve sparseness to some extent to allow expanding the group of subjects to include those subjects that may have no values for a few of the properties in the selected subset.

It may be noted that use of these matrices as materialized join views for performance gain needs to be evaluated against the workload for potential benefits versus the space overhead incurred for additional storage. The issue of which views to materialize is dependent upon the search pattern and it is up to the user to decide which is frequent search pattern.

The problems of obtaining property-specific statistics for a triple store with heterogeneous data can be mitigated with the use of statistics computed on the matrix materialized views because those can serve as statistics for the corresponding portions of the vertical table.

Finally, Jena2's [12] property tables (clustering multiple properties) are in many ways similar to subject-property matrices. The main differences include the following:

- Subject-property matrix is an auxiliary structure, not a primary storage structure. So, these matrices may be dropped or redefined as necessary without requiring a data re-loading.
- The definition of subject-property matrix allows use of nested properties and hence allows more ways of

creating useful materialized views for optimizing performance of a variety of queries in a workload.

Indexing Rulebases: Rulebases specified in RDF_MATCH query are applied, by default, during query processing to the specified list of models. However, if a rulebase is used frequently then that rulebase can be indexed using a set of APIs provided for this purpose. Indexing a rulebase for an RDF model refers to pre-computing the triples that can be inferred with respect to the specified model. These pre-computed triples are stored in a separate table and are used subsequently during RDF_MATCH query processing to speed up query execution. In general, a pre-computing may need to be done for a combination of models and rulebases, that is, applying a set of rules from the union of rulebases to a triples from the union of a set of RDF models.

However, these pre-computed results cannot be used directly to process RDF_MATCH queries that reference additional rulebases or models. Currently for such cases, all inferencing must be done at query execution time. Notice that inferencing can only add triples to the graph, so the pre-computed triples are always valid for the larger set of rulebases and models, though the pre-computed results are not necessarily complete. We plan to explore handling these cases by analyzing the rulebases and models so we can avoid re-computing portions of the pre-computed results that are complete.

Indexing RDF Data: As mentioned earlier, the core processing involves performing self-joins on IdTriples table. Thus, creating the right set of indexes on IdTriples is critical for performance improvement. There are typically two types of query patterns: 1) given a property, joining subject with subject, or object with object, and 2) given a property, joining subject with object, as shown below:

```
'(?r  ReviewerOf  ?c)
 (?r  Age         ?a)'
```

or

```
'(?r  ReviewerOf  ?c)
 (?c  rdf:type    Conference)'
```

Since property is typically specified as a URI value, index key with property as the first column may allow pruning the search space to a single range in the B-tree index. Further, having all the three columns (namely `PropertyID`, `SubjectID`, and `ObjectID`) as part of the key may allow index-only access provided the additional storage space required for three column indexes can be accommodated. Based upon these observations, we have used two three column indexes with the following keys in all of our performance experiments described in Section 5 : <`PropertyID`, `SubjectID`, `ObjectID`> and <`PropertyID`, `ObjectID`, `SubjectID`>. Use of key-prefix compression in indexes allowed reducing the storage space required for the indexes.

The choice for indexes may depend on the actual RDF data and workload characteristics. We need to explore further to see how any algorithm for choosing indexes may need to be customized to exploit constraints such as row formats used for RDF triples storage and typical RDF queries that involve multi-way self joins.

4. Minimizing Overheads by an Enhancement to RDBMS

This section discusses an enhancement to Oracle RDBMS table function infrastructure that can minimize table function processing overheads.

4.1 RDF Query Processing Components

The RDF query processing time using RDF_MATCH table function (t_{total}), without the kernel enhancement discussed in Section 4.2, can be represented as follows:

$$t_{total} = t_{core} + t_{sql2proc} + t_{proc2canonical} + t_{canonical2sql}$$

Here t_{core} represents the core processing time, that is, the cost of SQL query that performs the self-joins on IdTriples table and any additional joins with UriMap table. Once the results are computed, they are copied into variables of the table function procedure ($t_{sql2proc}$), and subsequently it is converted to canonical format ($t_{proc2canonical}$) so it can be returned to via RDBMS table function infrastructure, and finally transformed back ($t_{canonical2sql}$) so it can be consumed by the outer SQL query.

The component, $t_{total} - t_{core}$, is dependent on the result computed by table function (note: not on the overall result) and hence it will dominate the query costs when the table function result set size is large. The Experiment I (described in Section 5.3) demonstrates the overheads incurred for varying number of result rows. To avoid this overhead an enhancement to RDBMS is implemented as discussed below.

4.2 A New Table Function Interface

The following extension of RDBMS table function infrastructure is implemented, that would allow a simple rewrite of table function with a SQL query.

As an alternative to the current TableStart(), TableFetch(), and TableClose() interfaces, RDBMS should support a new table function interface:

```
TableRewriteSQL(arg1, …, argn) RETURNS VARCHAR;
```

This function takes the arguments specified in the table function and generates a SQL string. For table functions defined using this interface, RDBMS table function infrastructure does the following processing:

- Invoke the corresponding routine to generate the SQL string,
- Substitute the generated SQL string into the original SQL query, and
- Reparse and execute the resulting query.

The net effect is same as if the user typed in the generated SQL query in place of the table function. However, the general function mechanism cannot be used here because of the FROM clause. It has to be the table function.

Suppose the RDF_MATCH table function be defined using the TableRewriteSQL() interface. Consider the following query:

```
SELECT t.a age
FROM TABLE(RDF_MATCH(
  '(?r    Age    ?a)',
  RDFModels('reviewers'),
  NULL, NULL)) t
WHERE t.a < 25;
```

The resulting query after rewriting the table function is as follows:

```
SELECT t.a age
FROM (SELECT u1.UriValue a, u1.Type a$type
      FROM IdTriples t1, UriMap u1
      WHERE t1.PropertyID = 29 AND t1.ModelID = 1
            AND u1.UriID = t1.SubjectID) t
WHERE t.a < 25;
```

Note that the subquery in bold font is the SQL fragment that is returned from TableRewriteSQL() for the above RDF_MATCH invocation. Now, the whole SQL query is optimized and executed. For example, the filter condition is pushed inside the subquery for further optimization.

The advantage of such a scheme is that it avoids the overhead of copying the results into table function variables, as well as eliminates the table function infrastructure overhead of transforming the result to canonical form and re-transforming it back to present in the appropriate datatype format. However, such a scheme is applicable only when the table function can be defined declaratively using SQL (as is the case for RDF_MATCH).

5. Performance Study

This section describes the performance experiments conducted using RDF_MATCH table function.

5.1 Experimental Setup

The experiments are conducted using Oracle10*g* Release 1 (10.1.0.2.0) on a Red Hat Enterprise Linux AS 3 system with one 3.06GHz Pentium 4 CPU and 2048 MB of main memory. A database buffer cache of 256 MB, shared pool of 256 MB, and database block size of 8 KB is used.

The timings reported below are the mean result from ten or more trials with warm caches.

5.2 Dataset

The experiments I through IV are conducted using an RDF representation of WordNet [11], a lexical database for the English language, which organizes English words into synonym sets, categorizes these synonym sets according to part of speech (noun, verb, etc.), and enumerates linguistic relationships (antonymOf, similarTo, etc.) between these synonym sets. In the RDF representation, each part of speech is modeled as an rdfs:Class, and each linguistic relationship is modeled as an rdf:Property. This RDF Schema for WordNet is shown in Figure 3.

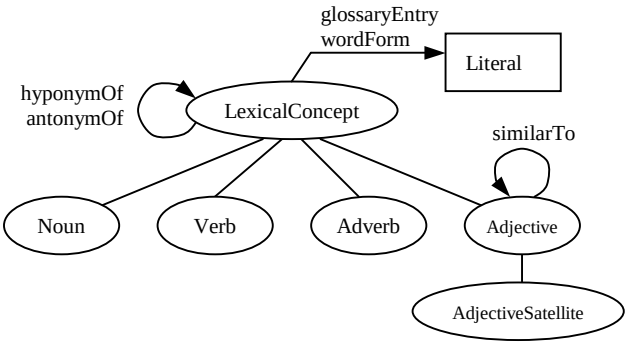


Figure 3: WordNet RDF Schema

The hyponymOf property is used to denote that the subject represents a specialization of the object. For example, skyscraper is a hyponym of building.

Table 3. Property and Resource Statistics of WordNet

Property	Count
WordForm	174,002
Rdf:type	99,653
glossaryEntry	99,642
hyponymOf	78,445
SimilarTo	21,858
Others	26
Total	**473,626**

Resources (explicit rdf:type)	Count
Verb	12,127
Noun	66,025
AdjectiveSatellite	10,912
Adjective	7,003
Adverb	3,575
Others	11
Total	**99,653**

The relevant logical statistics for the experimental configuration is shown in Table 3. The logical statistics can be computed simply with the RDF_MATCH table function. For example, to find number of resources typed as 'verb', a user use RDF_MATCH table function with the pattern `'(?w rdf:type wn:verb)'`. This type of query is expected to run efficiently as it results in a single table query. For example, the above query took less than 0.01 seconds.

The data is stored in the normalized form in two tables, namely, IdTriples table of size 14 MB and UriMap table of size 34 MB. The indexes on IdTriples table and UriMap table are of size 22 MB and 26 MB respectively.

Experiments V and VI use large-scale UniProt data with 80 million triples (see Section 5.7 for more details).

5.3 Experiment I: Overhead Estimation

This experiment characterizes the benefit of the TableRewriteSQL() enhancement described in Section 4. Four configurations are tested:

1) RDF_MATCH with the current table function interface (TableStart(), TableFetch(), and TableClose()). Execution time of this table function corresponds to the t_{total} term in Section 4.1.
2) SQL query equivalent to RDF_MATCH with the enhanced interface (TableRewriteSQL()). Execution

time of this query corresponds to the t_{core} term in Section 4.1.

3) Table function (using the current interface) that fetches from a SQL query, but does not return any rows. The SQL query is simple and its execution time is negligible. Execution time of this table function corresponds to the $t_{sql2proc}$ term in Section 4.1.
4) Table function (using the current interface) that returns rows, but does not execute any SQL. Execution time of this table function corresponds to $t_{proc2canonical} + t_{canonical2sql}$ in Section 4.1.

Figure 4 shows the query processing time for these components as the number of rows returned is varied: from the bottom, Core SQL, SQL to Proc, Proc to SQL, and Other in that order.

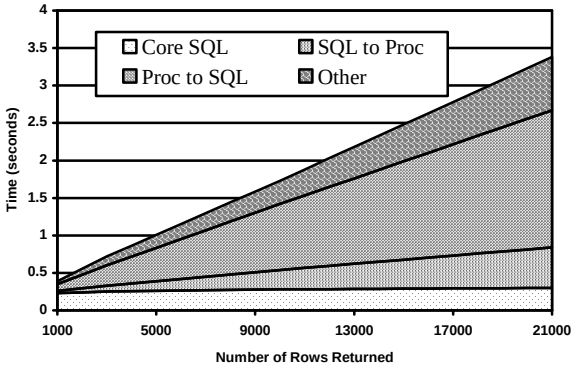


Figure 4: RDF_MATCH Query Processing Components (standard deviation $\sigma \leq 0.0838$)

The results demonstrate that $t_{sql2proc}$ and $t_{proc2canonical} + t_{canonical2sql}$ are linear in the number of rows returned, and that these overheads dominate the core SQL processing time when a large number of rows are returned. The enhanced table function interface avoids this per-row overhead, and therefore it is preferred over the current table function interface. In all of the remaining performance experiments, we run the queries with the enhanced RDF_MATCH table function interface.

5.4 Experiment II: Varying Number of Triples in the Search Pattern

As the number of triples in the RDF_MATCH search pattern increases, RDF_MATCH performs an increasing number of self-joins on the Triples table. To characterize how the varying number of self-joins impacts performance, queries are run to find 'hyponymOf' paths of varying length. For example, the query to find two-triple 'hyponymOf' paths is:

```
SELECT AVG(LENGTH(a))
FROM TABLE(RDF_MATCH(
   '(?a    wn:hyponymOf   ?b)
    (?b    wn:hyponymOf   ?c)',
  RDFModels('WordNet'),
  NULL, NULL));
```

The queries are run without materialized views, and with a generic `SubjectID-ObjectID` materialized view, as described in Section 3.3. Figure 5 shows the query processing time as the number of triples in the search pattern varies. Note that the number of matches declines as the number of triples increase, from 78,445 matches for the one-triple query to 45,619 matches for the six-triple query.

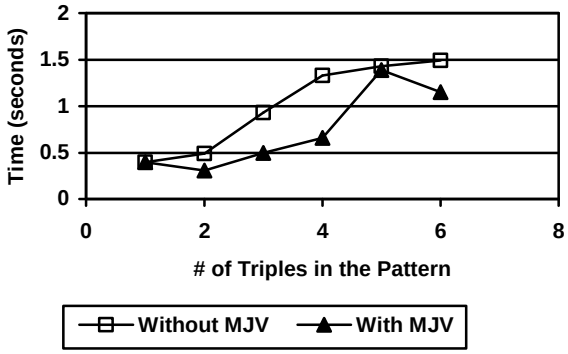


Figure 5: RDF_MATCH Performance For Various Searches ($\sigma \leq 0.0881$)

As expected, processing time increases with the number of triples due to corresponding increase in the number of self-joins. The materialized view generally improves performance, except for 1-triple and 5-triple case. For 1-triple case, no benefit is expected, as the resulting query does not involve any self-joins. For the 5-triple case, the benefit derived due to usage of materialized view is offset because the optimizer chooses a sub-optimal plan.

5.5 Experiment III: Varying Filter Conditions

This experiment characterizes the impact of SQL predicates that filter the results found by RDF_MATCH. The following search pattern is used for this experiment:

```
'(?c0    wn:wordForm    ?word)
 (?c0    wn:wordForm    ?syn1)
 (?c0    wn:wordForm    ?syn2)
 (?c1    wn:wordForm    ?syn1)
 (?c2    wn:wordForm    ?syn2)
 (?c1    rdf:type       wn:Adverb)
 (?c2    rdf:type       wn:Verb)'
```

This query is executed with four different equality filters (e.g., word = 'clear') and four different range filters (e.g., (word >= 'bat' AND word < 'bounce')) to yield approximately 350, 1050, 2000, and 3125 matches with each type of filter. Figure 6 shows the query processing time for these filters. Note that this query finds 79,885 matches in 8 seconds when there is no filter predicate. As expected, less selective filters require greater processing time. Notice that equality filters are more efficient than range filters. This is because the equality filter is implemented with a single lookup in the UriMap table to find the UriID for the literal given in the filter. In contrast, range predicates require a join between the IdTriples and UriMap table to get the values needed for filter evaluation.

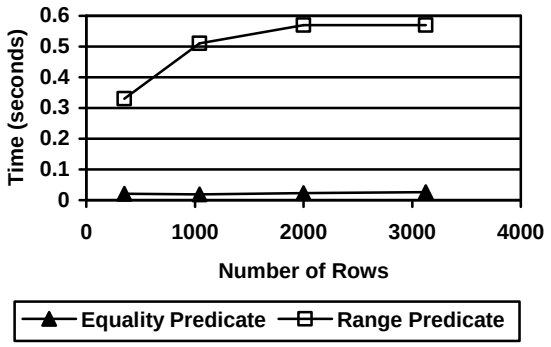


Figure 6: RDF_MATCH Performance with Filter Conditions (equality: σ ≤ 0.0029; range: σ ≤ 0.0619)

5.6 Experiment IV: Varying Projection List

This experiment characterizes the benefit of the projection list optimization done by RDF_MATCH. The following search pattern is used for this experiment:

```
'(?c0    wn:wordForm    ?word)
 (?c0    wn:wordForm    ?syn1)
 (?c1    wn:wordForm    ?syn1)
 (?c0    rdf:type       wn:Adverb)
 (?c1    rdf:type       wn:Adjective)'
```

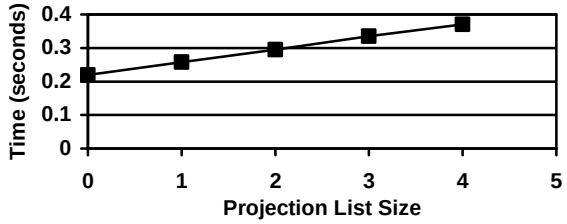


Figure 7: RDF_MATCH Performance For Varying Projection Lists (σ ≤ 0.0724)

This search pattern, which involves 4 variables and yields 1,470 matches, is used in queries with varying sets of variables referenced in the SELECT list. Figure 7 shows the query processing time as the projection lists are changed.

The projection list optimization eliminates joins with the UriMap table for variables that are not referenced outside of RDF_MATCH. It is clear that large performance gains are possible from this optimization.

5.7 Experiment V: Large-Scale RDF Data

This experiment characterizes RDF_MATCH performance for querying large-scale data. UniProt protein and annotation data in RDF format [14] is used for this experiment. To study scalability we created several datasets using varying subsets (from 10 million to 80 million triples) of the UniProt data. The largest dataset, corresponding to 5.2 GB of RDF/XML data, occupies 2.5 GB for IdTriples table, 1.7 GB for UriMap table, 3.6 GB for IdTriples indexes, and 1.2 GB for UriMap indexes. Six queries adapted from examples given with the UniProt data (shown in Table 4) are then run against these datasets.

Table 4. Queries adapted from UniProt sample queries

Description	Pattern	Projection	Result limit
Q1: Display the ranges of transmembrane regions	6 triples 5 vars	3 vars	15000 rows
Q2: List proteins with publications by authors with matching names	5 triples 5 vars 1 LIKE pred.	3 vars	10 rows
Q3: Count the number of times a publication by a specific author is cited	3 triples 2 vars	0 vars	32 rows
Q4: List resources that are related to proteins annotated with a specific keyword	3 triples 2 vars	1 var	3000 rows
Q5: List genes associated with human diseases	7 triples 5 vars	3 vars	750 rows
Q6: List recently modified entries	2 triples 2 vars 1 range pred.	2 vars	8000 rows

Each query includes a ROWNUM predicate to limit the number of result rows so that the number of matches remains constant even as the dataset size changes. Also, aggregate functions are used in the SELECT list to avoid the overhead of returning multiple rows to the client.

The RDF_MATCH search pattern for Query 1, for example, is as follows:

```
SELECT AVG(LENGTH(protein)), AVG(LENGTH(begin)),
       AVG(LENGTH(end))
FROM TABLE(RDF_MATCH(
   '(?p        rdf:type       up:Protein)
    (?p        up:annotation  ?a)
    (?a        rdf:type
               up:Transmembrane_Annotation)
    (?a        up:range       ?range)
    (?range    up:begin       ?begin)
    (?range    up:end         ?end)'
   RDFModels('UniProt'), NULL, NULL))
WHERE rownum <= 15000;
```

Execution times (in seconds) for these queries (see Table 5) remain almost the same even as dataset size changes.

Table 5. RDF_MATCH Performance Scalability

	Q1	Q2	Q3	Q4	Q5	Q6
10 M Triples	0.86	< 0.01	< 0.01	0.03	0.18	0.46
20 M Triples	0.95	< 0.01	< 0.01	0.03	0.19	0.47
40 M Triples	0.96	< 0.01	< 0.01	0.03	0.18	0.47
80 M Triples	1.03	< 0.01	< 0.01	0.03	0.20	0.49
Maximum σ	.054	0.002	0.002	.011	.065	0.07

This shows that RDF_MATCH based query performance is scalable, that is, retrieval cost per result row remains almost the same as the dataset size changes.

5.8 Experiment VI: Subject-Property MJVs

To see potential benefits from use of Subject-Property MJVs (SPMJVs), we used the following query pattern against the 80M triple UniProt dataset:

```
'(?s    up:name       ?n)
 (?s    rdf:type      up:Protein)
 (?s    up:curated    true)
 (?s    up:created    ?cre)
 (?s    up:modified   ?mod)'
```

An SPMJV was created for `rdf:type`, `up:curated`, `up:created`, and `up:modified` properties. This SPMJV contained 489,695 rows and occupied 39 MB; there was a single B$^+$tree index on the subject, which occupied 19 MB.

Two queries were tested: (#1) a COUNT(*) query, and (#2) a query that selects ?n, ?cre, and ?mod. Each query was posed with and without use of the SPMJV. The results in Table 6 shows that this can lead to significant performance benefits.

Table 6: RDF_MATCH Performance with and without SPMJVs

Query	Time (sec)
#1 w/o SPMJV	4.87
#1 w/ SPMJV	1.79
#2 w/o SPMJV	13.68
#2 w/ SPMJV	9.05

6. Conclusions and Future Work

The paper proposed a SQL based scheme for querying RDF data. Specifically, the RDF_MATCH table function is introduced with the ability to perform pattern-based match against RDF data (graph) that can optionally include triples inferred by applying RDFS or user-defined rules. Users can do further processing (iterate over, constrain using filter conditions, limit the results, etc.) using standard SQL constructs.

The RDF_MATCH table function itself is implemented by generating a SQL query against tables holding RDF data. For efficient query processing, generic and subject-property matrix materialized join views, and indexes (on RDF data and rulebases) are used. Furthermore, a kernel enhancement is implemented that eliminates RDF_MATCH table function run-time processing overheads.

The experimental study conducted using RDF data for WordNet and UniProt demonstrates that the SQL based scheme is efficient and scalable.

We expect that providing RDF querying capability as part of SQL will enable a database system to support wider range of applications as well as facilitate building semantically rich applications. The RDF querying capability can also be used in conjunction with data mining techniques on RDF data collected from diverse applications to discover interesting semantic relationships.

In future, we plan to consider alternate storage representations for RDF triples. A promising storage representation is partial normalization, where only the namespaces are normalized. That is, URIs are represented by the (namespace identifier, URI suffix). Also, we plan to enhance RDBMS optimizer to improve its capabilities in optimizing the class of self-join queries that typically occur while querying RDF data. The selection of suitable join method, join order, and subject-property matrix materialized join views is critical in generating an optimal plan. Allowing users to specify hints to influence the optimization process will also be explored.

Acknowledgments

We thank Jay Banerjee for his useful comments on earlier drafts of this paper.

References

[1] *RDF Primer*. W3C Recommendation, 10 February 2004, http://www.w3.org/TR/rdf-primer.

[2] L. Miller, A. Seaborne, A. Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. *First International Semantic Web Conference (ISWC2002)*, June 2002.

[3] *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation 10 February 2004, http://www.w3.org/TR/rdf-schema.

[4] T. Berners-Lee, J, Handler, O Lassila. The Semantic Web. *Scientific American*, May 2001.

[5] *RDF Data Access Use Cases and Requirements*. W3C Working Draft, 2 June 2004, http://www.w3.org/TR/rdf-dawg-uc/.

[6] Pipelined and Parallel Table Function. *Data Cartridge Developer's Guide Release 2* (9.2) Part No. A96595-01, Oracle Corporation, March 2002.

[7] R. G. Bello, et al. Materialized Views in Oracle. In *Proceedings of the 24th Int. Conf. on Very Large Data Bases*, (1998), 659-664.

[8] *WordNet, The Lexical Database for English Language*, http://www.cogsci.princeton.edu/~wn.

[9] *RDQL - A Query Language for RDF*, W3C Member Submission 9 January 2004, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109.

[10] *RDFQL Database Command Reference*, http://www.intellidimension.com/default.rsp?topic=/pages/rdfgateway/reference/db/default.rsp.

[11] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl. RQL: A Declarative Query Language for RDF. *WWW2002*, May 7-11, 2002, Honolulu, Hawaii, USA.

[12] Kevin Wilkinson, Craig Sayers, and Harumi Kuno. Efficient RDF Storage and Retrieval in Jena2. *First International Workshop on Semantic Web and Databases*, pp. 131-151, 2003.

[13] *SPARQL Query Language for RDF*, W3C Working Draft, 12 October 2004, http://www.w3.org/TR/2004/WD-rdf-sparql-query-20041012/.

[14] UniProt Data Set, http://www.isb-sib.ch/~ejain/rdf/.

[15] Li Ma, Zhong Su, Yue Pan, Li Zhang, Tao Liu. RStar: An RDF Storage and Querying System for Enterprise Resource Management. *CIKM*, pp. 484-491, 2004.